

PCT/IL 2004 / 00099 1
28 OCT 2004

REC'D 15 NOV 2004

WIPO PCT

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

October 05, 2004

**THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM
THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK
OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT
APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A
FILING DATE UNDER 35 USC 111.**

APPLICATION NUMBER: 60/515,664

FILING DATE: October 31, 2003

**PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH
RULE 17.1(a) OR (b)**

**By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS**



L. Edelen

**L. EDELEN
Certifying Officer**

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

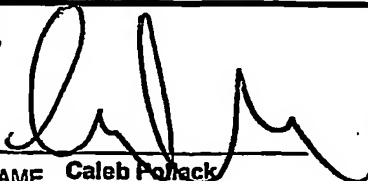
This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

INVENTOR(S)					
Given Name (first and middle [if any])		Family Name or Surname		Residence (City and either State or Foreign Country)	
HAGER		Yuval		Tel-Aviv, Israel	
RASAMAT		Emil		Tel Aviv, Israel	
LAN		Divon		Palo Alto, California	
ADDA		Michael		Jerusalem	
KIPNIS		Michael		Petach-Tikva, Israel	
<input type="checkbox"/> Additional inventors are being named on the ^ separately numbered sheets attached hereto					
TITLE OF THE INVENTION (280 characters max)					
DEVICE, SYSTEM AND METHOD FOR STORAGE AND ACCESS OF COMPUTER FILES					
Direct all correspondence to: CORRESPONDENCE ADDRESS					
<input checked="" type="checkbox"/> Customer Number		27130		Place Customer Number Bar Code Label here	
OR					
<input checked="" type="checkbox"/> Firm or Individual Name		Eltan, Pearl, Latzer & Cohen Zedek, LLP.			
Address		10 Rockefeller Plaza			
Address		Suite 1001			
City		New York		State	New York
Country		USA		ZIP	10020
		Telephone		212-632-3480	Fax
				212-632-3489	
ENCLOSED APPLICATION PARTS (check all that apply)					
<input checked="" type="checkbox"/> Specification		Number of Pages		85	<input type="checkbox"/> CD(s), Number
<input checked="" type="checkbox"/> Drawing(s)		Number of Sheets		2	
<input type="checkbox"/> Application Data Sheet. See 37 CFR 1.76		<input checked="" type="checkbox"/> Other (specify)		1. Appendix A (30 Pages) 2. Postcard	
METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT (check one)					
<input type="checkbox"/> Applicant claims small entity status. See 37 CFR 1.27.					
<input type="checkbox"/> A check or money order is enclosed to cover the filing fees					
<input checked="" type="checkbox"/> The Commissioner is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number:		05-0649		FILING FEE AMOUNT (\$)	
<input type="checkbox"/> Payment by credit card. Form PTO-2038 is attached.				80	
The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.					
<input checked="" type="checkbox"/> No.					
<input type="checkbox"/> Yes, the name of the U.S. Government agency and the Government contract number are:					

Respectfully submitted,

Date 31 / Oct / 2003

SIGNATURE



REGISTRATION NO.
(if appropriate)

37,912

TYPED or PRINTED NAME

Caleb Polack

TELEPHONE

212-632-3480

Docket Number:

P-6283-USP

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

This collection of information is required by 37 CFR 1.51. The information is used by the public to fill (and by the PTO to process) a provisional application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the complete provisional application to the PTO. Time will vary depending upon the

15535 U.S. PTO
60/515664



103103

United States Provisional Patent Application for

**DEVICE, SYSTEM AND METHOD FOR
STORAGE AND ACCESS OF COMPUTER FILES**

FIELD OF THE INVENTION

The present invention relates to data storage, data management and data access. More specifically, the present invention relates to devices, systems and methods for a version-controlled distributed filesystem.

BACKGROUND OF THE INVENTION

In some organizations, computer platforms may be located in various offices and branches, which may be physically separated by long distances. For example, a user may wish to use a first computer platform located in a first office, to access or modify a computer file stored on a second computer platform in a second office.

Some file systems may allow sharing of computer files over a Wide Area Network (WAN). An Enterprise File Server (EFS) may use a network filesystem, such as, for example, Common Internet File System (CIFS) or Network File System (NFS), to allow sharing of computer files over a WAN.

However, current solutions may suffer from bandwidth latency limitations and round-trip latency limitations. Furthermore, current solutions may suffer from various other problems associated with using a distributed filesystem when operating over a longer physical distance, for example, when operating over the Internet as a WAN.

DETAILED DESCRIPTION OF THE INVENTION

The subject matter regarded as the invention is particularly pointed out and distinctly claimed in the concluding portion of the specification. The invention, however, both as to organization and method of operation, together with objects, features and advantages thereof, may best be understood by reference to the following detailed description when read with the accompanied drawings.

It will be appreciated that for simplicity and clarity of illustration, elements shown in the figures have not necessarily been drawn to scale. For example, the dimensions of some of the elements may be exaggerated relative to other elements for clarity. Further, where considered appropriate, reference numerals may be repeated among the figures to indicate corresponding or analogous elements.

In the following description, various aspects of the invention will be described. For purposes of explanation, specific configurations and details are set forth in order to provide a thorough understanding of the invention. However, it will also be apparent to one skilled in the art that the invention may be practiced without the specific details presented herein. Furthermore, well known features may be omitted or simplified in order not to obscure the invention.

Some embodiments of the invention may use and/or incorporate methods, devices and/or systems as described in United States Patent Application Number 09/999,241, United States Patent Application Publication Number 2002/0161860, titled "Method and System for Differential Distributed Data File Storage, Management and Access", published on October 31, 2002, which is hereby fully incorporated by reference, and which is attached as "Appendix A" to this patent application. However, the scope of the present invention is not limited in this regard, and embodiments of the present invention may use and/or incorporate other suitable methods, devices and/or systems.

FIG. 1 (attached to this patent application) schematically illustrates a Wide Area Network (WAN) 1000 in accordance with some embodiments of the present invention. Network 100 may include, for example, an Enterprise File Server (EFS) 1001, a FilePort computer 1002, a FileCache computer 1003, and one or more client computers such as, for example, client computer 1004. Network 1000 may include various other suitable components and/or devices, which may be implemented using any suitable combination of hardware and/or software.

In some embodiments, EFS 1001 may include, for example, a physical file system 1011 and a filesystem server 1013. Physical file system 1011 may include, for example, a hard disk drive 1012 and/or another suitable storage unit or memory unit. Filesystem server 1013 may include, for example, a server utilizing Common Internet File System (CIFS) or Network File System (NFS).

FilePort 1002 may include, for example, a management unit 1021, a Distributed System File Server (DSFS) server 1022, a core server 1023, and a filesystem client 1024. Management unit 1021 may include, for example, components and/or sub-units as detailed below with reference to FIG. 2. DSFS server 1022 may include, for example, a computer able to perform a method of serving, creating, sending and/or transferring a data item, a file, a block or another suitable object in accordance with embodiments of the present invention. Core server 1023 may include, for example, a Windows NT server or a Linux server. Core server 1023 may include a cache 1025, which may include a suitable storage unit or memory unit. Filesystem client 1024 may include, for example, a client utilizing CIFS or NFS.

FileCache 1003 may include, for example, a management unit 1031, a file system server 1032, a core client 1033, and a DSFS client 1034. Management unit 1031 may include, for example, components and/or sub-units as detailed herein with reference to FIG. 2. Core client 1033 may include, for example, a Windows NT server or a Linux server. DSFS client may include, for example, a computer able to perform a method of requesting and/or receiving a data item, a file, a block or another suitable object in

accordance with embodiments of the present invention. Filesystem server 1032 may include, for example, a server utilizing CIFS or NFS.

Client computer 1004 may include, for example, a client application 1041 and a file system client 1042. Client application 1041 may include, for example, one or more suitable software applications, such as Microsoft Word, Adobe Acrobat, Adobe Photoshop, or the like. Filesystem client 1042 may include, for example, a client utilizing CIFS or NFS.

In some embodiments, file system client 1024 of FilePort 1002 and file system server 1013 of EFS 1001 may be able to communicate via link 1015, which may utilize, for example, CIFS or NFS. Similarly, file system client 1042 of client computer 1004 and file system server 1032 of FileCache 1003 may be able to communicate via link 1016, which may utilize, for example, CIFS or NFS. In some embodiments, DSFS server 1022 of FilePort 1002 and DSFS client 1034 of FileCache 1003 may be able to communicate via link 1017, which may utilize a method of distributed data transfer in accordance with embodiments of the present invention.

It is noted that links 1015, 1016 and/or 1017 may be wired and/or wireless, and may include, for example, one or more links which may be connected in serial connection and/or in parallel. In one embodiment, for example, links 1015 and 1016 may be Local Area Network (LAN) links, and link 1017 may include one or more links utilizing the Internet or other suitable global communication network.

FIG. 2 (attached to this patent application) schematically illustrates an exemplary embodiment of management unit 1021, which may include, for example, a web Graphic User Interface (GUI) 1051 that may be connected to a web server 1052, a Simple Network Management Protocol (SNMP) client 1053 that may be connected to a SNMP server 1054, a Common Language Infrastructure (CLI) 1055 that may be connected to a shell 1056, and a management Application Program Interface (API) 1057. Web server

1052, SNMP server 1054 and/or shell 1056 may be interconnected and/or connected to management API 1057, for example, using Remote Procedure Call (RPC) 1058.

Some embodiments of the present invention may minimize the amount of data that is transferred across network 1000. This may be achieved, for example, using a version controlled file system. In one embodiment, substantially each file or directory in network 1000 may have a version number associated with it. The version number may include a number that may increase with every change of the file or directory; and each version number may be associated with a specific version of a file or directory.

In some embodiments, client computer 1004 may require access to a file, denoted File1, which may be stored on EFS 1001. Client computer may request File1 from FileCache 1003, which in turn may request File1 from FilePort 1002, which in turn may request File1 from EFS 1001. In response, EFS 1001 may send File1 to FilePort 1002, which may store File1 and send it to FileCache 1003, which in turn may store File1 and send it to client computer 1001.

In one embodiment, each copy of File1 may have a Version Number (Vnum) associated with it. For example, FilePort 1002 and/or FileCache 1003 may maintain a cache of all or substantially all the files accessed during their operation, and a Version Number (Vnum) may be associated with substantially each file or directory saved in the cache.

When a first component of network 1000 needs to access a file, which may be stored on a second component of the network, the first component may send to the second component a file request and the Vnum of the file that is stored on the first component. If the second component has a stored file whose Vnum is the same as that of the file stored on the first component, then the second component may indicate so to the first component, and no further data transfer may be necessary. Alternatively, if the second component has a stored file whose Vnum greater than the Vnum of the file stored on the first component, then the second component may send to the first component data corresponding to the content difference (denoted herein: "Diff" or "delta") between the

two files, such that the first component may be able to rebuild the requested file from the Diff and the file stored on the first component.

For example, if there is a new version on a server, the server sends the delta between the latest version and the version that the client has cached. This may be performed, for example, using a suitable 'Diff Algorithm'. The delta between the two versions may be composed of one or more deltas, e.g., "patches", between the client's version and the latest version. The client may then apply the one or more patches, sequentially, to the file version in its cache, and update the Vnum to the latest version.

The server may save the files in its storage using a special format described below. Each file may consist of a base section, and a number of Diff sections. The base section may contain the data of the file, along with the base Vnum of the file. The base Vnum is the Vnum of the file if there are no Diff sections. Each Diff section added increases the Vnum of the file incrementally.

When a client requires to change data in a certain file, after verifying that the latest version of the file has been obtained, the client may send the difference (using a 'Diff Algorithm') between the latest version and the new version of the files being written by the client. The server then appends the Diff it received from the client to the latest version of the file, and incrementally increases the version number.

In some embodiments, whenever a file is changed, the clients that need to read the file from the server may read only the changes necessary.

In some embodiments, components of network 1000 may be physically located in various locations, branches and/or offices of an organization. For example, EFS 1001 and FilePort 1002 may be located in a headquarters office, a head office or a central office of an organization; EFS 1001 and FilePort 1002 may be located in physical proximity to each other, or may be connected to each other on the same LAN. In one embodiment,

EFS 1001 and FilePort 1002 may be implemented using one or more suitable software components and/or hardware components.

Similarly, in some embodiments, FileCache 1003 and client computer 1004 may be located in a remote office, a back office, or a branch office of an organization. For example, FileCache 1003 and client computer 1004 may be located in physical proximity to each other, or may be connected to each other on the same LAN. In one embodiment, FileCache 1003 and client computer 1004 may be implemented using one or more suitable software components and/or hardware components.

In some embodiments, FilePort 1002 and FileCache 1003 may be used to facilitate, speed-up or improve the transfer of data, files or blocks from EFS 1001 to computer client 1004, or vice versa. For example, FilePort 1002 and/or FileCache 1003 may store a copy of a file transferred through them or by them. Later, FilePort 1002 and/or FileCache 1003 may be requested to transfer a file or to obtain a file, for example, on behalf of computer client 1004. In some cases, FilePort 1002 and/or FileCache 1003 may detect that the requested file has not been changed since it was last stored in the cache of FilePort 1002 and/or FileCache 1003. The requested file may be sent to computer client 1004 from FilePort 1002 and/or FileCache 1003, thus saving an access to EFS 1004.

In some embodiments, FilePort 1002 and/or FileCache 1003 may compare the Vnum, the content and/or a property of a requested file, to a corresponding Vnum, content and/or property of the requested file which is stored on EFS 1000. FilePort 1002 and/or FileCache 1003 may otherwise analyze and/or compare files, blocks, directories and/or traffic passing through FilePort 1002 and/or FileCache 1003, to detect that a requested file is identical or non-identical to a corresponding file stored in the cache of FilePort 1002 and/or FileCache 1003.

In some embodiments, the analysis or comparison may further allow FilePort 1002 and/or FileCache 1003 to calculate, compute and/or produce a "Diff" or "delta" portion, which

may include data indicating the modifications that need to be done to a first file in order to create a second file.

In one embodiment, a first device may request a file from a second device. The first device or the second device, or a third device in the network, may detect that the first device has stored a relatively old version of the requested file. Upon such detection, instead of sending the entire new version of the requested file to the first device, only the "Diff" or "delta" portion between the new version and the old version may be sent to the first device. Then, the first device may rebuild or reconstruct the newer version of the file based on the older version stored in the first device and the "Diff" or "delta" portion received.

General Description of an Exemplary System

W-NAS™ is a new storage technology category that enables storage access from great geographical ranges (over WAN), be they inter-city or inter-continental.

W-NAS™ may provide the first commercial implementation of a means of centralizing storage as well as sharing files across a distributed enterprise (e.g., companies with more than one site).

FileCache™ appliance is deployed at each remote site, and a FilePort™ appliance is installed at the central data center(s). All desktop computers throughout the enterprise store data directly on the enterprise-wide file servers at the data center(s). There are no infrastructure changes required in order to install the system, and no software needs to be installed on the file servers or remote site workstations.

Some embodiments may include one or more appliances that are deployed at the edges of the enterprise network; for example, at every branch office that wishes to access storage or grant access to storage, over the WAN. These appliances may be plug-and-play devices; in some embodiments, no infrastructure changes are required, very little needs to

be configured in order to set them up, and no software needs to be installed on the file servers or remote site workstations.

Exemplary Components Description

At a remote branch office, an appliance named W-NAS FileCache™ may be installed. This appliance appears on the remote site's LAN as a regular Windows® file server. However, rather than serving files from its own hard-drive (as a regular file server does), it utilizes the DSFS™ protocol technology, in order to fetch the files from the data center file server, over the WAN, in an efficient way. To do this, the FileCache™ connects over a TCP/IP channel to a second appliance, called the W-NAS FilePort™, installed at the corporate data center. When receiving a request from the FileCache™, the FilePort™ turns to the actual file server, acting as a Windows client on behalf of the actual user that originated the request, and gets the needed information. The appliances are transparent to the end-users, which use the same tools they are accustomed to when accessing Windows file servers.

The whole system may be managed from the center using a dedicated management station, using a web browser (each appliance also has an individual web interface).

Both the center and the edge can be deployed using a no-single-point-of-failure architecture in order to achieve high availability.

The W-NAS™ technology architecture provides for a many-to-many relationship, which means that a single FilePort™ can serve many remote sites, each with its own FileCache™, and that a single FileCache™ at a remote site can access data through multiple FilePort™ devices, each at a potentially different data center.

Exemplary Functional Description

In some embodiments, W-NAS architecture may be based on sophisticated file system protocol tunneling. A FileCache™ is placed in each remote office requiring access to files residing at another site (the enterprise data center). This FileCache™ looks to the clients on the remote office network as a regular file server residing on that network. In fact, the FileCache™ receives requests from the remote office clients as a regular file server would do, but rather than serving these requests from its local hard disk, it tunnels them over the WAN using the DSFS™ protocol, to the FilePort™ that resides at the data center. The FilePort™, receiving the requests tunneled from the FileCache™, acts as a regular client when accessing the data center's file server in order to fulfill the original client's request.

In an effort to address the bandwidth and network latency problems, W-NAS may use intensive algorithmic optimizations, both on the FileCache™ and the FilePort™, in order to reduce both the amount of data sent over the network (addressing the bandwidth problem), and the number of round trips needed between the FileCache™ and the FilePort™ in order to service a client's request (addressing the network latency problem).

The WNAS system may be based on a version-based differential distributed network file system protocol (DSFS™).

Description of Exemplary Operation

In some embodiments, when a file is requested to be read, or written onto, it undergoes several layers of optimizations and modules of the system. The purpose of those layers is to serve as much as possible from the local cache, without hurting semantics, and if the server needs to be contacted, it should be in the most efficient way.

In exemplary embodiments of the invention, some of the layers may include the following:

1. Local Storage

Some files are known to be less important to the administrator, or they appear for a short time and then disappear. In some embodiments, the system chooses to leave those files at the edge, and perform all the operations locally there, without sending them back to the center.

2. Local Caching

In some embodiments, each part of file that is being read by the client is saved locally at the edge, for the case it will be needed again. If the second request for the same data was within a short period of time from the first, it is served directly from the cache. If some time has passed, it is verified with the center that this is the correct version, and then it is served from the cache. One of the most important design goals of the system is that every full set of data is sent over the wire only once, and after that only deltas are sent.

3. Version-based protocol

In some embodiments, each file is assigned a version number. Files may be cached at various places along the route (on the client, FileCache™, FilePort™, or main memory of the file server itself). The DSFS™ system contains cache-coherency mechanisms that keep track of what version of the file is cached in each location, and uses this information to minimize traffic. For example, if the up-to-date version of a file requested by a client happens to be cached on the FileCache™, there is no need for the FileCache™ to request that file from the FilePort™. Similarly, if an older version of a file requested by a client is cached on the FileCache™, then only the delta ("diff"), i.e. the actual bytes that have changed between the cached version and the up-to-date version, needs to be fetched from the FilePort™ to the FileCache™.

4. Application-level optimizations

In some embodiments, as the FileCache™ acts as if it were a file server on the remote office's local network, it is aware of every file-system I/O request coming from applications. The FileCache is able to detect request patterns and based on these patterns,

perform highly sophisticated optimizations, that further reduce WAN traffic between the FileCache™ and the FilePort™.

5. Diffing

In some embodiments, an independent algorithm for computing binary Diff on two files may be deployed. It should be a sufficiently fast, yet efficient algorithm, so it will detect changes that were made to the file, even if an unknown binary format is used. Changes could be of several forms, such as insertions, deletions, block moving etc.

6. Speculative Diffing

Certain applications write different files to the file system although they contain similar data. In some embodiments, a method to identify which files might contain the same data is used, and it is applied to the DSFS™ protocol, in order to negotiate deltas of different files.

7. Compression

In some embodiments, in addition to all of the above, all data sent through the wire undergoes standard compression (such as Lempel-Ziv) in order to further reduce the amount of traffic.

8. Prefetching

In some embodiments, since most of the branches access a pre-defined set of data, it can be pre-fetched periodically to the cache, to make sure data is fresh, and no additional transaction are needed during the day. This helps increase the cache hit rate to close to 99%, and increase and improve user experience

9. Read ahead

Different files call for different access patterns. In some embodiments, the system learns the way applications use certain files, and try to fetch the relevant records of the file before the user requests them (if they are not there already).

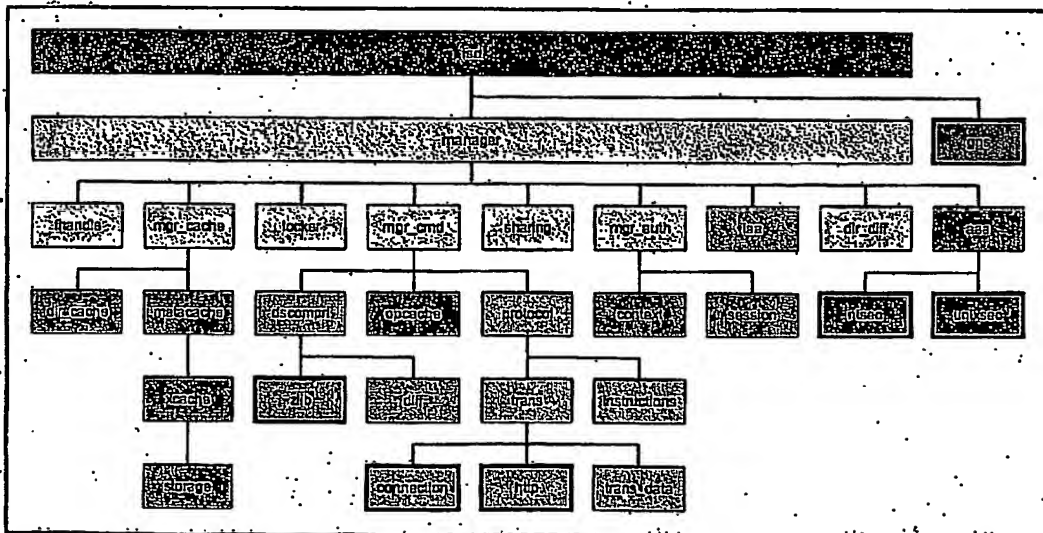
10. Write on close

In some embodiments, a write is being delayed until the file closed, or until a significant amount of data is waiting to be committed to the file. This enables to reduce the number of transactions to the file server, and save on bandwidth. It does not affect file system semantics, since CIFS/NFS does not define a write to block until data is written to disk.

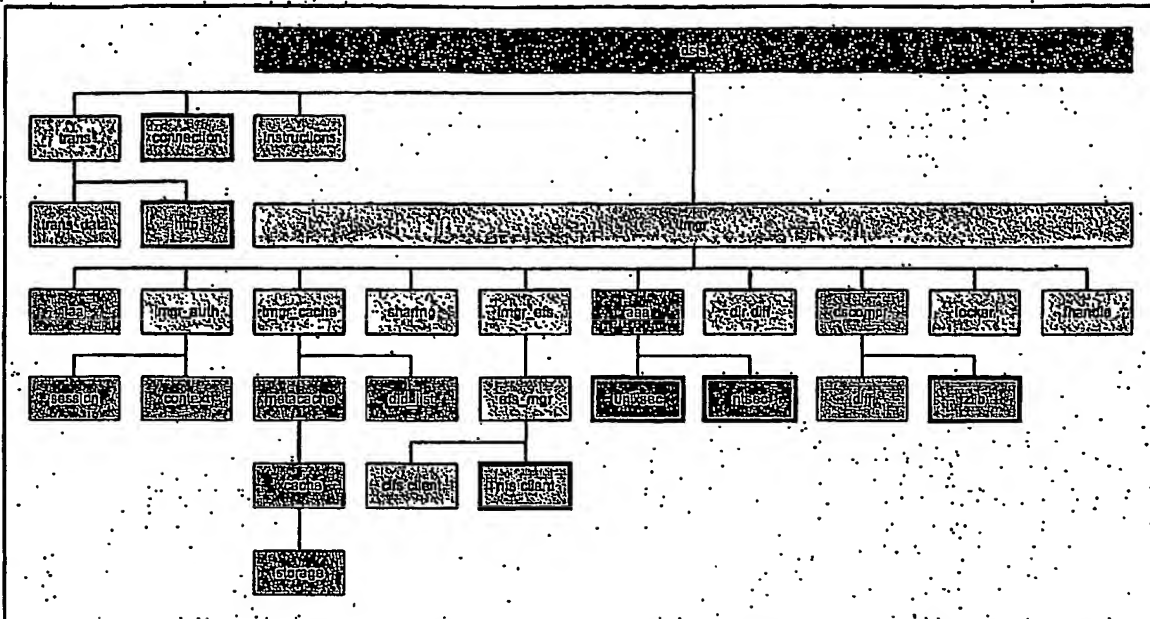
Modules

The following Figures describe exemplary embodiments of internal module structures of FileCache and FilePort engines according to embodiments of the invention.

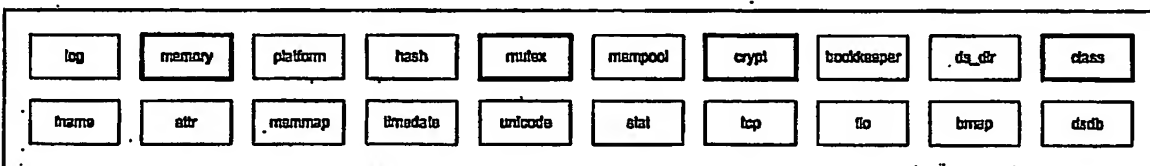
DSFS™ client engine (FileCache)



DSFS™ server engine (FilePort)



DSFS™ infrastructure modules



The legend:

- | | | |
|-------------------------------------------|-------------------------------------------|-----------------------------|
| OS upwards | Authorization authentication and auditing | Compression and diff |
| FileCache transparency towards the client | System's layout | Infrastructure/OSAL modules |
| DSFS™ core | Transport | Local appliance cache |
| Existing modules are in bold | | |

Exemplary Features

Some embodiments may include one or more of the following features and functionalities:

1. Synchronous, reliable "store" operations

In some embodiments, when a user hits an application's "Save" button, that application will receive an OK status and continue its operations, only when the data is safely saved on the data center file server. In other words, the FileCache™ does not deploy "store and forward" logic. This is an important characteristic needed to achieve reliable storage. If something goes wrong (for example, the user is out of disk quota or the file server is down), the user should receive a notification of this event, and be given the opportunity to save his data elsewhere.

2. Fast "store" operations

In some embodiments, the requirement of reliable "store" operations poses a challenge regarding the performance of application "save" operations. The W-NAS™ system addresses this by drastically reducing the amount of data that needs to be sent over the WAN in order to complete a successful "save" operation. This is achieved by a combination of advanced compression techniques, differential transfer (sending only the bytes that have changed), and application-level optimizations (see below).

3. WAN-wide file sharing

In some embodiments, DSFS™, being a synchronous protocol, enables file-sharing semantics with full distributed locking across the WAN. For example, a typical application would allow the first user opening a document to be granted full read-write access to that document, and would lock the document for the period it is open. Subsequent users concurrently attempting to open that document would be granted read-only access. This standard LAN behavior is supported by DSFS™ over the WAN.

4. Security

In some embodiments, DSFS™ fully supports native operating system security mechanisms. For instance, in the Windows (CIFS/SMB) environment, full access control (ACL) permissions are enforced and native authentication is supported both for Windows NT version 4 (Domain Controller) and for Windows 2000 (Active Directory). For network security, DSFS™ deploys internal measures, such as session-key based message digital signing. In addition, DSFS™ supports and relies on any network security mechanisms already installed on the network such as Firewalls and VPNs. DSFS™ is designed to operate over TCP/IP port 80, thus there is no need to open an additional port on the Firewall. It is also important to note that all user sessions are path-thru all the way, which means that the server believes that the real user is accessing it directly (instead of through our system). This has other implications such as simple auditing, quota management, and owner preservation.

5. Many-to-many architecture

In some embodiments, a FilePort™ installed at a data center can access many file servers. In some embodiments, a FilePort™ may be concurrently accessed by many FileCache™ devices at many remote offices. In some embodiments, a FileCache™ can access many FilePort™ devices.

6. Internationalization

In some embodiments, DSFS™ supports the Unicode standard and is designed to allow a single installation of a DiskSites system to work across languages and time zones.

7. Supported application

In some embodiments, DSFS™ may be used with "document processing" applications. A description of such an application is: applications that have a concept of a "file" or "document" which the user works on, and then saves. Common applications of this type include Microsoft Office® applications, graphic designing applications, software and hardware engineering application, etc.

8. Central Management

In some embodiments, the system can be managed as one object using a central management station. It enables the administrator to deploy defined policies on groups of appliances, and monitor the group altogether.

9. Transparency to the user

In some embodiments, the user that works with the system doesn't see it as a different external system. The FileCache appears on the local network as if it were the central server (it can even have the same name), so from the user's point of view, he is accessing the central file server – as if it were on his local LAN. In some embodiments, no training or client's installation is needed in order to start working with the system.

10. File Server transparency

In some embodiments, the FilePort utilizes CIFS protocol as client (and NFS). Since this is standard protocol, it is agnostic to which server is installed at the customer site. Be it a Windows 98 machine, Windows 2K, Solaris running samba, Novell with CIFS support, Network Appliance filer, EMC Celler or HDS SAN – they are all supported seamlessly. Moreover, the system also supports heterogeneous systems.

11. High availability

In some embodiments, both the FileCache and the FilePort can be installed in High Availability mode. The software supports it, and only the hardware is required to deploy a NSPF (No Single Point of Failure) solution.

12. Other Features

Embodiments of the present invention may include one or more of the following features and/or functionalities:

Core Technology

File Streaming support

Disconnected operation (read only)

Product additions

Print services

QoS Management

Installation

High availability at the center

High availability at the remote branches

Seamless upgrades

Management

XML-RPC API

Device web interface

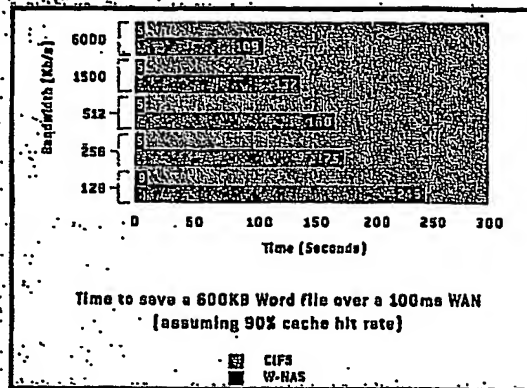
Central management engine

Central Web Management

Performance

We have measured the time it takes a system according to an exemplary embodiment of the invention to save a 600KB word file over a 100ms WAN line, with a few different bandwidth ratios. As can be seen from the graph below, one embodiment maintains a constant 5 second time, while working naively with CIFS, it takes anywhere between 2 to 4 minutes. It can also be seen that adding a lot of bandwidth (such as, quadrupling it) does not help a lot, since the main problem with CIFS is latency.

As can be seen, embodiments of the present invention may solve this problem.



Functional Features

Some embodiments of the invention provide a WAN file system that enables true file storage consolidation. This may be achieved by the complete replacement of local file servers with FileCache appliances. By centralizing the storage, the organization achieves reduction of IT costs, an ability to maintain and backup data centrally and greatly enhanced data security. Some embodiments may include one or more of the following features: near LAN performance, synchronous operation, full file system semantics support, reliable data transport, and environment-based system management. Some embodiments may include one or more of the following functional features:

1. Synchronous WAN file system

In some embodiments, the file system is synchronous, which means that client requests are completed only upon their completion on the central file server. One embodiment is only a transport system and never stores the user's critical data. This architecture enables full support for file sharing semantics. Since the system is synchronous, it requires high responsiveness, which in turn requires a highly aggressive, predictive, and sophisticated set of optimizations on transfer of files, both data and meta-data.

2. Block-based WAN file system

In some embodiments, enterprise systems may involve various types of files. Office files are usually in the order of tens of megabytes, while other files may reach hundreds of megabytes or more. A WAN file system may be required to provide file size-independent performance. Therefore, the smallest independent caching unit is not a file but rather a file block. In some embodiments, block-based caching may include and/or use block handling such as block-based versioning, block-based diffing, block-based compression, and block-based management. In one embodiment, since cache sometimes cannot be trusted, the various cases of blocks that were deleted from the cache need to be handled transparently.

3. High-performance WAN file system

In some embodiments, to achieve high performance over the WAN, a large set of optimizations for data and meta-data transfer may be used. Optimizations include: Save-as identification (ability to relate different files by their name/context/work pattern); Speculative resemblance (ability to relate files that are different objects but contain similar or identical data); Predictive read (expect blocks that are about to be read by the user/application and read them in advance (requiring careful analysis of application and user behavior); Compression; Diff (fast and effective ability to calculate a binary difference between two files/blocks), Versioning (each block snapshot is given a unique version number, and only deltas between versions are transferred on the network – both ways), Content-based caching (blocks that belong to different files are stored only one time in the cache).

4. Content-based metadata - independent WAN file system

Our research has revealed that many different files that belong to different users share the same data. Some embodiments may use this knowledge to save storage for caching, and/or to improve performance by substantially never fetching again a block that we have already fetched once. This feature may be fully transparent to the file system users, who believe that different files contain different information. A decision algorithm is used for when a block can be written to and when a copy should be created.

5. Centralized policy based resource management

In some embodiments, to manage such a large system, a customer needs central management functionality. This may require: (1) An API for each individual device on the network; (2) A Web interface for each individual device on the network; (3) A central management station to enable the management of groups of devices. Central management is implemented by applying certain policies (like cache configuration, security, pre-fetching definitions, etc.) on a predefined group of appliances. Policies should be applied to all appliances at once and errors reported in a clear way. If an appliance has a different configuration from the group, it should be noted clearly in the interface. Queries on the configuration of a group should be handled in the same way. Information should be collected and aggregated in a human readable format. Resources should be managed across components to ensure high service level to the user.

6. Environment based caching

In some embodiments, the system is aware of the customers' potentially different environmental conditions. Therefore, a large set of options are provided to configure the behavior of the general system. An administrator can define per-share parameters such as branch exclusiveness (only one branch can change the files and there is no need to lock on the center, to check cache validity, etc.); read-only (files can never be written to, which can help optimization and allow some applications to open files for write although they do not intend to write to them); read-all (no security checks and no need to read ACL from the server or to parse them along the way); caching priorities (some files may be more important than others, and in some cases one might want to make sure that they stay longer in the cache); change-frequency (some shares change more frequently than others, which can be used to tune the amount of transactions used for cache validity verification).

7. High availability WAN file system

In some embodiments, when installed at a large customer site, the system may require high availability functionality, which means that two or more appliances should back up

each other and cover up in case of failure. This has different implications on the remote branch and on the center. Moreover, the implementation may be active-active, meaning that the stand-by machines are not idle but used to serve user requests. High availability may require special attention due to the file based nature of the application because data needs to be maintained in a way that system failure will not cause corruption or data loss.

In some embodiments, issues such as management of the cluster as one machine, installation, upgrades, virtual IP addresses, leader election, and so on may need to be handled carefully.

8. Rule-based file system tuning engine

In some embodiments, the system is implemented as an engine that provides the basic functionality with a superimposed static rule set. The rules can then be changed on the fly by field engineer, QA team, etc. This also makes possible different pricing models and the ability to quickly fine tune the system to meet customer needs.

9. Support for multiple writers over the WAN

In some embodiments, the system supports multiple writers to the same file while maintaining performance, a feature based on smart record locking.

10. Cache coherence issue

In some embodiments, the latest written data should always be read, so that the cache is used smartly and file/block versioning is sufficiently sophisticated not to corrupt the data corruption while maintaining high performance.

11. Consolidation of Novell shares over the WAN (Pass through Authentication)

Novell 5.1 or later has an add-on to support CIFS. Unfortunately, it does not support GetFileSecurity CIFS transactions, which makes it difficult for caching technologies because there is no security information about the file. To overcome this issue, in some embodiments, we send all operations pass-through to the file server and learn, in time, what were the results of each security request (operation caching). When the user requests an operation on a file he had requested before, he receives the same response if it

is within a valid time. As in every caching module, significant attention needs to be paid to situations of cache invalidation.

Additional Technological Features

1. Solving the problem of Windows™ CIFS timeouts

We discovered that CIFS may not behave well under some embodiments of the system because the time required for certain operations is too long (since the work is synchronous). To overcome CIFS timeouts, we may use a method of returning data to the CIFS client before we obtain the full response, and fool it into believing that the file system indeed responds within the specified time limit. This should happen regardless of network conditions (bandwidth, congestion) because the time constraint is a hard limit.

2. Version numbers for blocks and/or files

In some embodiments, to implement the sending of file and block deltas over the WAN, a sophisticated method of tracking version numbers of blocks and files is used. The solution is to track, both at the FileCache and at the FilePort, the changes made internally (i.e., by self) that increase only the version number of the specific block, and the changes made externally that increase the vnum of the entire file and therefore of all the blocks in it (given that there is no information about the exact place where the change was made in the file). In some embodiments, all this should happen in $O(1)$, as it is not acceptable to change all the information for all the blocks for every such change.

3. Aggregated file system instructions with internal dependencies

In some embodiments, to reduce the amount of transactions over the WAN, intelligence for aggregating file system operations may be used. There are at least two types of aggregation: predictive and piggybacking.

In some embodiments, predictive aggregation is used when the system expects a specific transaction and "holds" the previous transaction (if possible as a result of synchronous operation semantics) to see whether there is another transaction on the way. One example

is deleting a directory, which translates into a GetFileAttributes and DeleteFile for each file in the directory tree.

In some embodiments, Piggybacking is performed when a certain operation forces a transaction and is added to several other transactions that were on hold (like write dirty blocks), or when it is expected that several transactions will be required at a later stage (like get directory attributes, read ahead transactions).

4. Delta based, directory transport and caching

In some embodiments, whenever a directory is changed (file metadata), we need to send only the records that represent the files that were changed. In some embodiments, since we do not have hooks to identify what was really changed, an algorithm is used to compare a cached directory with the real one. The result should be file IDs that were changed. Such a change could be a delete, rename, write, change attribute, create, etc. In some embodiments, only this information is sent over the wire, which is then reassembled at the other end.

5. Smart LRU caching

In some embodiments, all the cache management functionality may use a sophisticated storage subsystem that knows how to store files quickly, without disrupting the data path, and can fetch them quickly, regardless of the size of the database. All this should run in kernel mode, so the option of using other technologies does not exist. In some embodiments, each file is given an ID along with its version number, which creates a unique reference to it in the file system. Priority queues may be managed, in order to remove the right files when the cache is full.

6. Smart cache pre-population

Some embodiments may include a method to synchronize the cache (usually at night). Instead of automatically fetching each file and checking versioning information, a set of block and block versions is sent to the central FilePort, which then responds with fresh

information about the files (metadata and data). This may be optimized to network conditions and load.

7. File system operations pattern recognition

In some cases, a WAN file system cannot maintain performance without identifying similar sets of data. Many modern applications do not open a file and write to it but move it to different folders under different names, writing to a different file, etc. Users also maintain different versions of files, usually by renaming them. The difference between the data in these files is often minor. In some embodiments, we apply sophisticated behavior pattern matching algorithms to identify these similarities and exploit them when sending data over the WAN.

8. Enhanced automatic resource balancing per device

In some embodiments, since the system uses local resources to save on remote resources, there are some cases and conditions (extensive load, high-bandwidth networks, low latency, etc.) when we may make a decision whether to run the algorithms and try to save bandwidth, or send the data over the network as is (since running the algorithms takes time, it may end up getting there faster). The algorithm may consider the dynamic aspects of the system: current load, current network status (latency, packets drop, and congestion), file and storage types, and user priority.

Additional Features

Some embodiments may include one or more of the following additional features:

1. Delta calculation between two binary files. Some embodiments enable to discover a difference in $O(n)$ complexity.
2. File size independent performance. Ability to avoid CIFS timeouts.

3. Full file system semantics support – including quota, authorization and file sharing.

4. File system operations pattern recognition.

5. Ensuring data availability and consistency over the WAN.

6. Full network file system transparency to the end user (authentication).

7. The CIFS (previously SMB) and NFS protocols have been around for two decades. They have achieved wide-spread acceptance and serve as key infrastructure components in any computerized enterprise environment. However, over this time the protocols have not been adapted to WAN environments. Therefore, as long as there is no major change in the underlying protocols (a task difficult to achieve, given the huge user base that will require an upgrade) - a solution such as our W-NAS will be needed in any place where users will want to use standard environments and applications over the WAN.

8. In some embodiments, the product may be an "off the shelf" product in general availability. However, the product may be designed and built specifically with the intent to provide the ability for Value Added Resellers (VARs) and Integrators to extend the system with customer specific adaptations.

9. In some embodiments, the product may operate in a file storage environment, which may require a central file server (NAS or SAN with a file-system front-end) to function. The product is compatible with most wide spread NAS/SAN vendors' offerings such as Network Appliance, EMC, Hitachi Data Systems, Microsoft Server/Advanced Server/Data center Server.

Other Key Elements of Some Exemplary Embodiments

1. High performance, Synchronous, block-based WAN file system - (File streaming Support)

Some embodiments implement the core system functionality that enables work on files over the WAN, using a synchronous file system. The file system is block-based, and is highly optimized to provide high performance (near-LAN) over wide area networks. This task involves various issues like: solving the problem of CIFS timeouts, blocks versioning, Aggregated file system instructions, block-based in memory diff, directory diffing, smart LRU caching, smart fetching, save-as (pattern identification), Undiffable, and others.

2. Disconnected operation (read only)

Some embodiments may include support for doing read-only operations on the remote cache when the WAN connection is down. This involves methodologies on how to behave when partial information is available, such as ACL's, and how to handle authentication without an authentication server, and without compromising on security more than needed.

3. High availability and Installation

Some embodiments may implement a pair-wise, active-active high availability solution. Each FilePort / FileCache may be installed as a pair of machines, that will run two instances of the FilePort software. In case of a failure, the surviving machine will take over the failing instance. Instance migration will be possible due to standard techniques: shared storage (SCSI or SAN), serial heartbeat, resource fencing (STONITH). In addition to the standard issues, we may handle cases of data that was not written to the disk at the time of the failure, both at the FilePort side, and at the FileCache side.

4. XML-RPC API

In some embodiments, an XML-RPC implementation may be used in order to provide system API.

5. Central management engine

Some embodiments may handle groups of appliances, and manage policy instead of configuration.

6. SNMP v3 support, especially SNMP authentication

7. Logging facilities

8. Asynchronous operation

Some embodiments may split the synchronous engine to an asynchronous one. This may include management of a state between requests and responses, and also the ability to return with approximate answers to the user. It may also involve management of the data - since data may reside at different locations in the system.

9. Rule based optimization modules per application

Some embodiments may divide the system to a generic WAN file system engine, and activation rules based on application and usage patterns. This may involve using a general, extensive and powerful engine. It will also give more flexibility in the product definition, pricing model and add ability to fix problems in the field at zero time.

10. Caching prediction

Some embodiments may study different file types and different application behavior and make sure the system reads ahead files data before the user requests it, to save time.

11. CPU / Network tradeoffs

Some embodiments may include, devise and/or implement an algorithm that will compute, at each point in time, the fastest path to the user data. It can decide on maximum compression, or none at all, enlarge priorities etc.

12. Mail Services

Some embodiments may integrate some of the solutions that exist in the market for mail and calendar collaboration -- to a complete product, that will support our system. In one embodiment, the system may include Print services. For example, one embodiment may integrate print queues management (such as CUPS and/or SAMBA) into the product, and add management interface to it, so the system may supply print queue management.

13. Environment based caching

Some embodiments may enable maximum performance by fine tuning the system according to environment conditions, such as: exclusive shares, read only shares, read all shares, caching priorities, share change frequency.

14. Pass Through Authentication

In order to support Novell's Native File Access, some embodiments may implement pass-through authentication. Pass-Through authentication (PTA) is used to delegate security enforcement responsibility to the CIFS server at the center. The CIFS server validates the user credentials with the Domain Controller and only then grants the user access to a resource on the CIFS server. A clear advantage of the above authentication method is full ACLs support including file owner preservation, access rights, permissions hierarchy without any changes of existing users, groups and domain security settings and full quota support. In addition, such system may support normal file system auditing functionality.

Version Controlled File System

Some embodiments may use a Version Controlled File System.

For example, one embodiment may minimize the amount of data that transfers across the network. One of the main methods to achieve that goal is working with a version controlled file system.

Each file or directory have a version number associated with it. The version number is a number that with every change of the file, increases, and each version number is associated with a specific version of a file.

DSFS clients and DSFS cache servers maintain a cache of all the files accessed during their entire operation. The cache may be maintained on any suitable storage media (Disk, memory etc.). A version number is associated with each file or directory saved in the cache.

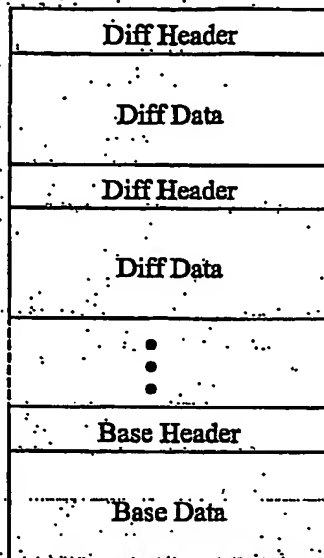
In some embodiments, whenever a client or cache server needs to check for file updates (see 'Cache Coherence' for further details about when an update is necessary), it sends to the server the Version Number ('Vnum') that it has on cache. If there is a new version on the server, the server sends the delta between the latest version, and the version that the client has cached. For details about computing the delta between files versions, see 'Diff Algorithm'. The delta is composed of all the deltas created between the client's version and the latest version. The client then applies the patches of the file, serially, to the file it has cached, and updates the vnum to the latest version.

In some embodiments, the server maintains the files, in its storage, in a special format described below. Each file is comprised of a base section, and a number of diff sections. The base section contains the data of the file, along with the base vnum of the file. The base vnum is the vnum of the file, if there are no diff sections. Each diff section added increments the Vnum of the file.

In some embodiments, whenever a client wants to change the data of a certain file, after making sure it has the latest version of the file, it sends the difference (see 'Diff Algorithm') between the latest version, and the new version of the files it tries to write. The server then appends the Diff it received from the client to the file, and implicitly increments the version number.

In some embodiments, whenever a file is changed ONLY the clients that need to read the file from the server read ONLY the changes necessary.

Figure 3: The file structure, as appears on the server

**Speculative Diffs**

Many applications that access files for reading and writing use complex methods for accessing the files. Applications often do that to be able to restore from catastrophes, backup purposes or others, as the application programmers see fit.

From a file system point of view, the aforementioned operations are a set of different operations on different files. Some embodiments of the invention, however, may allow to understand a full set of operations as a set related to one file, and by being able to identify different files as similar (or, in our set or notions – different versions of the same file), it can run the Diff Algorithm (see 'Diff Algorithm'), and send only parts of the data.

In some embodiments, the identification of the patterns or the sets of operation, is done due to a state machine that is managed both by the server side and by the client side. The transitions between the different states are the operations the application performs on the files. Whenever there is a situation that two different files are to be "diffed", only the diff is sent over the network, and both the server and the client know to which file to relate the diff to.

In some embodiments, the following scenarios are identified by the system, and instead of sending the full file data, in a naive implementation, a diff is computed, and sent over the network (if the files are not similar -- the full file data is sent).

All these scenarios start with file X that exists in the system:

Scenario A.

- a new file with name X is created, replacing the original file
- new data is written to the newly created file X.

Scenario B.

- file X is deleted.
- a new file named X is created.
- new data is written to the newly created file X.

Scenario C.

- file X is renamed to Y
- a new file named X is created
- new data is written to the newly created file X
- file Y is deleted

Scenario D.

- file X is renamed to Y
- a new file Z is created
- new data is written to the newly created file Z
- file Z is renamed to X
- file Y is deleted

Other scenarios might occur also.

In some embodiments, the implementation may be as follows:

On the client side:

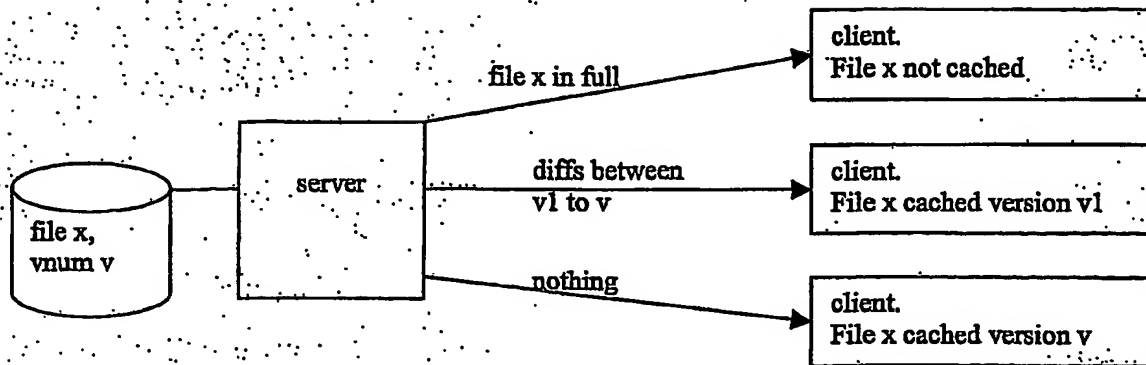
Whenever a file is deleted, renamed or replaced, a copy of the original file is saved and marked – 'lost'. Whenever a new file is created and data is written, a search is done on all the lost files to see if there is a file there that is similar (see 'Diff Algorithm'), and if there is, this file id is sent to the server, along with the diff data itself.

On the server side:

Whenever a file is deleted, renamed or replaced, a copy of the original file is saved and marked 'lost'. Whenever the client sends a diff, and marks to the server to use a file that was "lost", the lost file is used as a base, and the diff is placed upon it.

In some embodiments, the system is designed in a way that if one of the components – server or client – does not have one of the lost files in its storage, the lost file will not be used.

Figure 4: A server with clients



File System Connection Management

A WAN file system server may need to support very large number of users. Therefore, any resources used, should be used at a minimum, if possible. In great amounts of users, TCP/IP connections may become a very large resource consumer. Some embodiments of the invention may include a connection management mechanism to address that issue.

In some embodiments, all critical communication between the client and the server are done over the TCP/IP protocol. The basic idea here is closing the TCP/IP connection when it is not needed, while being able to tell when the client is still active or not. The difficulty here is that clients might lock files – and when the connection is closed, we do not know if the client is still working, or gone forever.

In some embodiments, the solution may include the following:

Whenever a client connects to the server, they do a hand-shaking, and decide about a session key for the session. They also have a session id associated with the session, and the server has a file describing this session on a special file (session context).

The TCP connection has a relation of many-to-one with the session. There can be any number, including zero, of TCP connections for one session.

- When a client connects to the server, it creates a TCP/IP connection.
- If the client does not perform any operation over the connection for a certain amount of time – the server closes the connections and marks it as a connection with no active TCP connection (CTX_WAIT_RECONNECT). The client identifies the socket closure – but does nothing.
- When the user tries to perform an operation on the remote file system, the client, transparently creates a new TCP connection with the server and sends the operation. The operation data include the session-id, and the server can find the context, and mark it as CTX_ALIVE again.

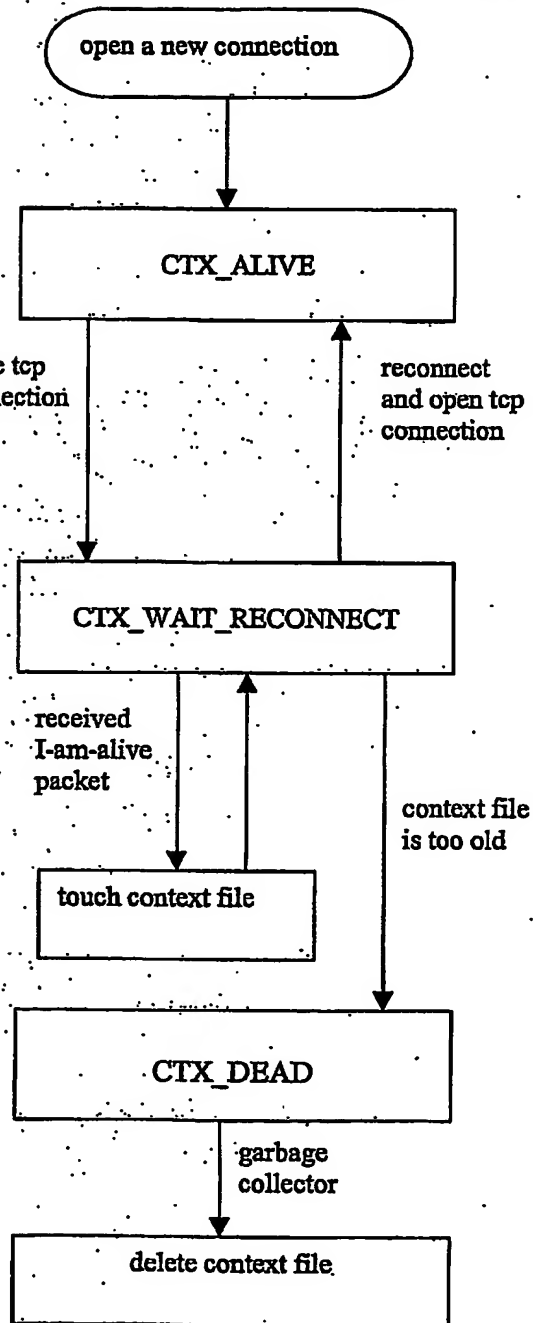
During the time that the session is in CTX_WAIT_RECONNECT state, if the client has a file opened for write, it sends to the server every IAA_SERVER_TIMEOUT/6 seconds, a UDP packet, stating that it is still alive (an "I-am-alive" packet). The context file is touched (i.e. its modification time is set to the current time) every time an I-am-alive packet arrives.

In some embodiments, if the client returns on-line, it can continue to work. If another client connects and tries to change a file, that is locked, the following may happen:

- If the file is locked by a connection, that it's context is marked CTX_ALIVE, the lock request will be denied.
- If the file is locked by a connection that it's context is marked CTX_WAIT_RECONNECT, and it's modification time was less than IAA_SERVER_TIMEOUT seconds ago, the lock request will be denied.
- If the file is locked by a connection that it's context is marked CTX_WAIT_RECONNECT, and it's modification time is more than IAA_SERVER_TIMEOUT seconds, the lock request will be granted – and the file will be marked as locked by the new connection.
- If the file is locked by a connection that is marked as CTX_DEAD, the lock request is granted, and the file is marked as locked by the new connection.
- If the file is not locked, the lock request is granted, and the file is marked as locked by the new connection.

In some embodiments, this mechanism may enable us to keep resources usage to the minimum, while smartly locking and unlocking files by different client.

Figure 5: Context State Machine



File-System Tunneling

The widely-used network file system protocols are CIFS in the Windows® world and NFS in the Unix world. These protocols usually come as standard features in their respective environments.

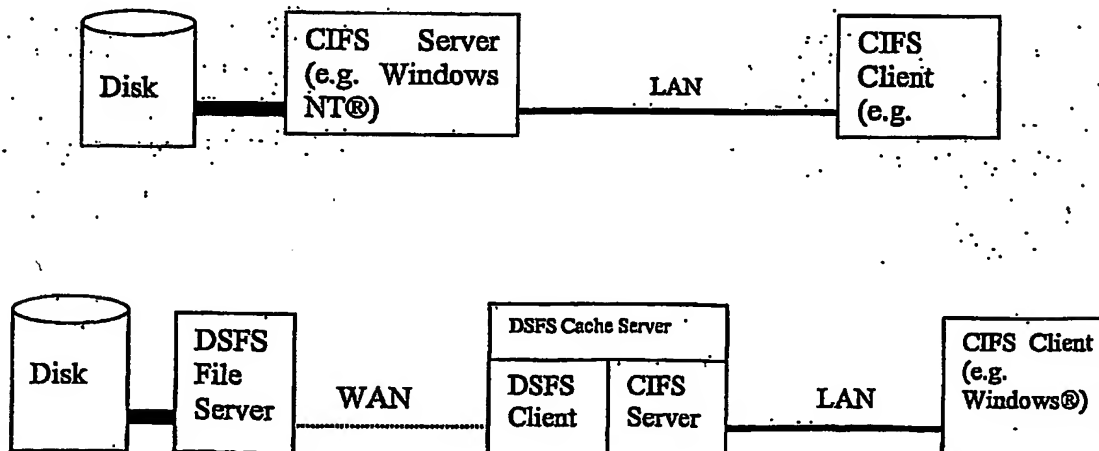
Therefore, in order not to install software on the client side, some embodiments may use a gateway (also known as the DSFS Cache Server) that is a CIFS (or NFS) server on the one hand, but a DSFS client on the other. In other words, when acting as a CIFS server and receiving a request from a CIFS client (e.g. a Windows® computer on the LAN), rather than accessing the local hard-drive in order to find the data needed to satisfy the request, the Cache Server turns to the DSFS File Server, using the DSFS protocol to fetch the data.

We call this behaviour "tunneling" CIFS through DSFS.

Illustration:

Normal CIFS operation

(e.g. on a Windows NT® server): (we use CIFS as an example, could be NFS).



1. CIFS Client sends a request to the CIFS server (e.g. "Read File").
2. CIFS Server receives the request and read or writes required info to/from the attached disk.
3. CIFS Server sends a response to the CIFS client.

CIFS tunneling through DSFS:

1. CIFS Client sends a request (e.g. "Read File") to the "CIFS server" side of the *DSFS Cache Server*.
2. *DSFS Cache Server*, acting as a DSFS Client, processes the request and sends it over the WAN using the *DSFS Protocol* to the *DSFS File Server*.
3. *DSFS File Server* receives the request and read or writes required info to/from the attached disk.
4. *DSFS Cache Server*, acting as a DSFS Client, receives the *DSFS File Server's* response, processes it, and sends it as a CIFS response to the CIFS client.

1. Exemplary System Overview

1.1. System Architecture

In some embodiments, we chose the following architecture for the system: The FileCache™ is installed on a standard PC, over Linux operating system. Versions may vary. In some embodiments, we have used successfully kernel version 2.2.16, 2.2.19, 2.4.18 and 2.4.20. We have used RedHat™ versions 7.0, 7.3 and 9.0 successfully (however, other suitable versions may be used).

In some embodiments, we run a slightly modified version of Samba™ 3.0.0 on the FileCache™ (user mode application). In some embodiments, modifications include: 1) removed support for

batch opportunistic locks; 2) added support for sharing mode (exists only in Windows™ and not Unix™ environments); 3) additional hooks for measurements of statistics.

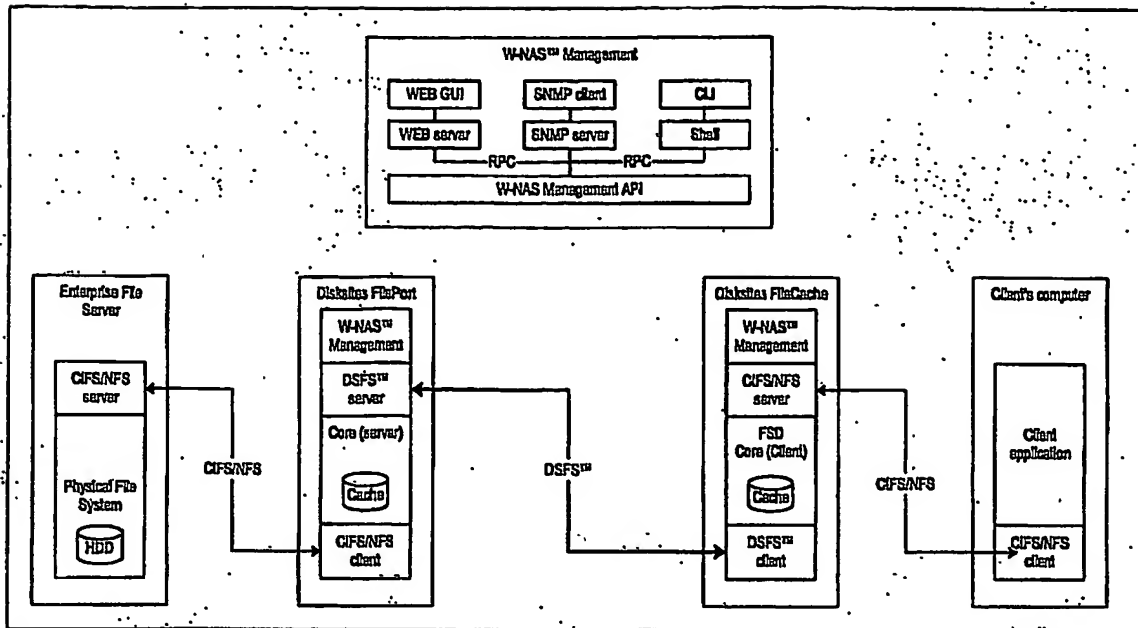
In some embodiments, we are running the most (or at least some) of the FileCache™ code as a standard Linux™ kernel file system. Both NFS server and the Samba™ server use our file system (DiskSites File System – DSFS) as a standard Linux file system.

In some embodiments, we chose to implement substantially all of the system calls inside the kernel mode, using, for example, standard kernel API, instead of using a user mode agent – for example, for performance and debugging simplicity.

In some embodiments, communication is done over a standard TCP/IP channel (Except I-am-alive requests that may use UDP communication).

In some embodiments, FilePort™ is running in user-mode as a whole, and uses TCP/IP communication with the EFS. In some embodiments, for CIFS client, we have implemented ourselves; and for NFS client, we may mount the NFS share on the server on our appliance, and use standard file system calls (for example, for simplicity).

In some embodiments, one can implement NFS client himself, and have the freedom to enjoy fine tuned access to the protocol parameters; in some embodiments, this may be unnecessary.



1.2. System layout

In some embodiments, a FileCache™ provides access from multiple users on multiple clients and may access multiple FilePorts™. A FilePort™ provides access from multiple FileCaches™ and may access multiple EFSs. In order to provide such many-to-many layout we introduce a concept of contexts and session entities.

In some embodiments, context defines the logical link between one FileCache™ and one FilePort™ so that one context is defined, for example, by a pair of ids. These ids are unique (system wide) and are factory or deployment generated. Each peer (FileCache™ or FilePort™) holds a list of valid contexts. In this design FileCache™ periodically sends one or more "I am alive" datagrams (or signals, packets, frames or messages) to all FilePorts™ that exist in its contexts list, for example, to validate its contexts on their side.

In some embodiments, session defines the CIFS/NFS session between the user and the EFS. Session, being tunneled via a WAN file system is substantially always served through the same peer of FileCache™_FilePort™ and, therefore, belongs to the certain context. When context dies (substantially for any or no reason) all its sessions are destroyed on both sides.

In some embodiments, for example, using Branch Level Security, FileCaches™ may create one session per one FilePort™-EFS link. This session may belong to the specially defined branch user.

1.3. Automatic resource tuning

This is a description of a WAN file system engine that uses its resources to the optimum, and in some embodiments does so dynamically and/or automatically. Note that both ends of the system, although having different tasks, may be designed in substantially the same way or in a similar way.

In some embodiments, the system may include the following components: A file system engine; a data collector; a decision algorithm.

In some embodiments, the engine does substantially all the file system operations, and the data collector gathers information about the way the engine works. The decision algorithm chooses the best way to perform a certain operation, according to the data collected.

Below is a more detailed description of the way the system works in some embodiments:

1. A file system engine

- i) Serve file system requests.
- ii) Compress (or encode) and decompress (or decode) data.
- iii) Calculate the difference between files, and/or patch files.
- iv) Handle several users/files/sessions substantially at once.

2. A data collector

- i) Available bandwidth and roundtrip latency.
- ii) Available CPU/memory resources.
- iii) Compression effort (CPU/memory/time) and ratio.
- iv) Diff effort (CPU/memory/time) and ratio.
- v) Static data: user/application priorities.
- vi) Response times from the server.
- vii) Static data: required service level, according to user/application
- viii) Cache miss ratio

3. A decision algorithm

In some embodiments, when a request arrives, the decision algorithm may try to anticipate the effort and gain in each route of operation, and may decide which may be the best mode (or substantially a better mode, or a relatively better mode) to serve the user. For example, in some embodiments:

- i) If compressing a file and sending it to the other side may take more than just sending it – don't compress.
- ii) If sending a diff may have a high risk of hitting a cache miss on the other side (which may require to re-send the whole data) – don't send a diff in the first place.
- iii) If a higher priority user is currently using the CPU, don't run diff/compressions.
- iv) If the required service level for the user/application could not be achieved – notify the administrator.
- v) If the application allows, it can choose to work asynchronously in order to achieve the requested service level.

2. System Features

2.1. Block based engine

In some embodiments, in order to optimize the traffic over the WAN, the internal cache handling and the delta calculation on the appliances in accordance with embodiments of the invention, we may divide a file or files into one or more blocks. In some embodiments, these blocks are the minimal data unit for transport and caching, and may be either of constant or variable size.

In some embodiments, we may use, for example, constant size blocks (such as 128KB), because we believe there is no much gain to performance, but many additions to the code complexity and stability. However, one may use other suitable block sizes and/or dynamic variable size blocks.

In some embodiments, using blocks, the FileCache™ gets from the FilePort™ substantially only those blocks that may contain the data that was requested by the clients, and sends back substantially only those that have changed.

In addition, in some embodiments, since the FileCache™ utilizes application-based read-ahead prediction (see Read-ahead and write-back predictions further in this document), it may request a certain block of a file, without being bothered with actual file size; if the prediction was untapped

— we only lost the single block treatment. On the other hand, when several FileCaches™ are working with the same FilePort™ on the same file, the block based system enables to refine delta exchange, so that the FilePort™ can notify its FileCaches™ which block has changed exactly.

In some embodiments, underlying layers of Windows clients software (such as the CIFS client) may have a non-configurable timeout, and the file-system operations like Open, Read, Write, Close and Move may not overpass. In some embodiments, this timeout may be very short and varies between 60-180 seconds depending on the OS version (this limitation defines the maximal possible block size as such that can be sent during 60-180 seconds under the specific network conditions. A cautious administrator may define the maximal block size as half of it, so there will be no troubles at peak times). Reading/Writing a large file in a whole may bring to this timeout in certain network conditions. This is another important impact of working with blocks: in some embodiments, the system is file-size independent.

In some embodiments, implementing the block-based WAN file system may require handling and/or implementing the following:

(i) Version management for blocks of files

In some embodiments, the whole system may be based on version management. Substantially each block and file has a version number attached to it at any point in time. Whenever a file is changed, the version number changes. When a FileCache™ requests a file, it also adds to the request information about which version it has cached. If the file in the EFS is different, The FilePort™ sends an update in a form of a delta from that version.

In some embodiments, managing versions for files and blocks may require some sophistication — especially since the files might change without the system knowing about it (i.e. a user changes a file directly on the EFS). Another pre-requirement may be scalability — some embodiments of the system should mark and discover the changes of even huge files (those are files of hundreds and thousands of megabytes) in $O(1)$ complexity, without a need to update or check all file's blocks.

In some embodiments, a versioning mechanism may be used to manage versions. Below is a short description on how it works in an exemplary embodiment. The following method may be used by both the FileCache™ and the FilePort™. Both entities may deal with receiving requests for data and either responding from the cache, or forwarding to the next entity. Therefore, the file and

block versioning mechanism may be exactly the same (or substantially the same, or substantially similar). This fact may simplify the design and implementation.

In some embodiments, since substantially each block is stored in the cache and transmitted separately, substantially each block has its own version number. In addition, in order to distinguish between different versions of files, each file has its own version number.

In some embodiments, substantially each file stored has a pair of numbers that compose the version number (vState): *internal* and *external*.

In some embodiments, internal version number is the last version number of the opened file that was changed by the current entity.

In some embodiments, external version number is the last known version number of the file (changed either by the current or a different entity).

In some embodiments, blocks whose version number is between internal and external version number of the file are treated as valid.

In some embodiments, the following operations may be used:

Open a file – if the file was changed at the next entity, increase its external and internal version number.

Read a block – If the block is valid, read from the cache. If it is stale (or not valid), ask the next entity for a new block, and update its internal and external version numbers.

Write a block – Change the internal version number of the block and send a diff (or full file, see *diff calculation*) to the next entity.

(ii) Dirty/plain blocks concept.

In some embodiments, FileCache™ tries not to block the clients on their write requests, rather storing the changed *dirty* blocks isolated from their previous known version called *plain*. In some embodiments, when the file is closed, the plain cache holds last known file's data and metadata.

FileCache™ substantially always uses *local* block version for READ/WRITE operations. This *local* may be either the *plain* or the *dirty*.

In some embodiments, the following rules may apply to manipulating with *dirty* and *plain* blocks and metadata on FileCache™.

1. Retrieving the *local* version for READ:

Check if *dirty* version exists. If so, return *local* = *dirty*.

Check if the block is a "zero" one. If so, create *plain*, fill it by zeroes (see change file size below).

Otherwise, if *plain* is missing or expired in cache, get it from FILEPORT™.

Return *local* = *plain*.

2. Retrieving the *local* version for WRITE:

If *dirty* is missing, retrieve *local* for read (see the above paragraph), and create a *dirty* copy of it. Note: In some embodiments, that's why we are substantially always adding AR to AW during the OPEN. Set *dirty* size due to the known file's size.

Return *local* = *dirty*.

3. During last block's READ (either *local*=*dirty* or *local*=*plain*), the system may limit itself due to the file size.

4. During SET_FILE_SIZE the size of the old last block's *dirty* is updated.

5. During commit, WRITE instructions are issued substantially only for the blocks that have *dirty*.

6. File size reduction may cause immediate commit.

7. During commit, if the file size has changed, a SET_FILE_SIZE instruction is added first.

8. After commit, the FileCache™ replaces *plain* with *dirty* – both file data (blocks) and metadata. If it was no *dirty*, and the block size has been changed, the system just changes *plain* size, if needed.

9. When the file is closed, the plain cache on FilePort holds last known file's data and metadata.

In some embodiments, FilePort™ writes data synchronously, so that it does not manage *dirty* on its side. Instead, it needs to handle a diffs collection per each block.

In some embodiments, one or more or all of the following rules may apply to manipulating with file blocks and metadata on the FilePort™:

1. During the READ/WRITE, before updating the *plain*, the system checks whether a block is a "zero". If so, creates a zero *plain*.
2. During the SET_FILE_SIZE, a new *plain* for old last block is generated and a diff is created and stored. The system does not create/delete the *plain* and *diff* blocks that have been added or disappeared as a result of SET_FILE_SIZE.
3. When file's metadata is generated for the very first time, it gets a default *bmap*.

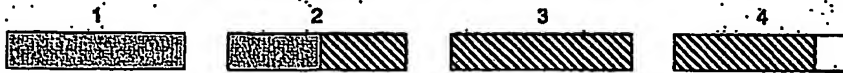
(iii) Change file size

In some embodiments, increasing a file size for a system that manages and caches blocks may be completed in $O(1)$ time, no matter how many blocks were added/removed from the file. In some embodiments, there is a way to "mark" blocks as "new" and thus all of their contents is zero, or "old" and thus can be discarded by the cache mechanism. In some embodiments, due to this approach, both FileCache™ and FilePort™ may manage in file's metadata a bit mask called *bmap*.

In some embodiments, the *bmap* holds for each file's block whether it is "zero" or not. When the file is created, its *bmap* is empty. When the file is being reduced or enlarged, its *bmap* is being reduced or enlarged accordingly. Newly added blocks become zeroed. When the block is written, a zero mark is cleared. Default *bmap* substantially always contains all non-zero blocks.

The following is an example which may be used in one embodiment:

The file has been enlarged; blocks (3) and (4) have been added (however, neither *plain* nor *dirty* are created at this time). If *dirty* of block (2) exists, it is enlarged and the delta is filled with zeroes. *Bmap* is enlarged accordingly; all newly added blocks are signed as "zero" blocks. File gets a "size changed" mark. FilePort™ will be notified during the next commit.



The file has been truncated; blocks (3) and (4) have been removed (However, only superfluous *dirty* blocks are deleted; *plain* blocks remain in cache for future diff usage). If *dirty* of block (2) exists, it is truncated. *Bmap* is reduced accordingly. File gets a "size changed" mark. FilePort™ is notified immediately with the commit. After commit, if it was no *dirty* for block (2), its *plain* is truncated and stored with a new version number.

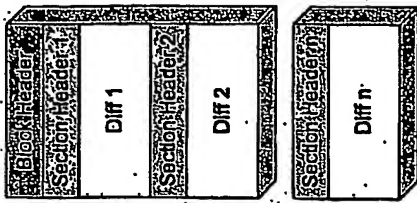


In some embodiments, another way to store "zero" information, may include a list of pairs, for example, <latest stale version number #, starting from block#>. This map may be defined with a constant size – for example, 20 entries. When it comes to be overflowed, the list is truncated with "last version number+1, 0" pair. Old version numbers that could be believed will be lost – FileCache™ will issue a transaction to FilePort™. This list may become a part of vState of file's metadata.

(iv) Management the collection of diffs

In some embodiments, FileCache™ and/or FilePort™ can (substantially always) reconstruct a last file version from the block_{base-version} and the collection of [diff_{base version+1}.. diff_{last version}]. (See Delta (diff) calculation between blocks above). In some embodiments, FileCache™ substantially always initiates the requests, so it is can manage (but doesn't have to manage) old diffs in its cache. In some embodiments, FilePort™, on the other hand, may manage these diffs to support multiple FileCaches™ - each one with its own block versions.

In some embodiments, the diffs may be stored per block in regular LRUEd cache and may be of the following format:



(v) Content based storage

Some embodiments may use a different approach to implement a system to cache and transfer blocks of files. In some embodiments, blocks are only referred to by a hash on their content (hash may be any suitable hash algorithm, for example, MD5). Blocks are treated as never changing. The blocks are saved in the disk in a way that enables fast access according to the block hash (an example of such implementation could be to save all the blocks in a special directory, and make sure the file name is the block hash).

In some embodiments, for each file, we will not have a list of blocks, rather a list of block hashes. When a file changes, we do not change the block itself, but use a different block, that is stored on a different hash. This way, we only cache and transfer each block once over the network -- if many files share similar blocks, we may use this similarity, for example, to save bandwidth and cache space.

In some embodiments, when the FileCache™ needs to read a block, it sends the FilePort™ the block number in the file, and the hash result, and if it is cached or not. The FilePort™ checks the latest version of the file at the BFS. If the FileCache™ has the right hash in the right place, nothing needs to be done besides sending an approval to the FileCache™. If the FileCache™ doesn't have the right hash (for example, if the file has changed after the FileCache™ read it), the FilePort™ needs to send an update.

The FilePort™ can send the update in one or more suitable ways, for example:

- 1) It can send only the new hash, without the data, hoping the FileCache™ has the new block cached from some other file. If the FileCache™ doesn't have it cached, it notifies the FilePort™ and asks it to send the full data (or a diff);

2) It can send the new block as a whole. The FileCache™ might already have the block cached, and thus can ignore the data received;

3) It can send a diff between the new block and the old one.

In some embodiments, the decision on which action to take may be based, for example, on one or more of the following conditions or criteria:

- If the FileCache™ doesn't have the original block, no diff will be sent
- If the FileCache™ recently notified the FilePort™ that he has the new block cached, only block hash should be sent
- If the latency is high, only the block data should be sent
- If bandwidth is low, only the block hash should be sent
- If many files hold references to that block, only the block hash should be sent
- In any case we are sending the block data, we may try first to run the diff. If CPU is not available or bandwidth is very high, this may be avoided.

In some embodiments, the information above needs to be collected and normalized. It may be fine tuned or optimized to the customer's environment.

In some embodiments, FilePort™ also manages a database (in the file system) of deltas between different hashes. In one embodiment, this may be done, for example, by storing the delta files with names of the format <source-hash>-<dest-hash>. This way, a once computed delta will be stored, and if needed again, sent without re-computing it. All storage of blocks, hashes and deltas may be managed by the LRU cache. In case a block is missing, it may be re-read from the EFS; in case that a delta is missing, it may be re-computed.

2.2. Multiple writers support

In some embodiments, some applications are intended for multiple users to work on the same file. An example is database applications, but there are others too. Applications of that sort may have the risk of reading or writing garbage data, since there might be another application doing the opposite operation on the file.

In some embodiments, there are a few methods for synchronization that can prevent such results, for example:

- 1) Do nothing. Maybe the environment will ensure that each instance is working on different locations in the file, maybe you will implement a mechanism to identify these problem and overcome them;
- 2) Proprietary synchronization. The instances of the application might synchronize on a proprietary protocol – it can be a direct protocol, using a third entity (so called "manager"), or using the file system (i.e. the first to create a shadow file wins);
- 3) Using file system locks. In some embodiments, this may be the standard way. An application that wants to work on a portion of a file, may lock that portion for that operation and may release it later. Other instances may need to check for locking, or may be denied to interfere by the server.

Some embodiments may support such applications, for example, by using and/or implementing the following:

Write only what was written – When writing data to the EFS, we may make sure that exactly what the user wrote to the FileCache™ is written to the EFS. That also includes the fact that he might have written data that is exactly the same as the data that was there before – the mere fact that he wrote it is important. When the user writes information to the FileCache™, the FileCache™ records the ranges in which data was written. The FileCache™ computes the delta from the previous version, and sends over to the FilePort™, along with the ranges list. The FilePort™ rebuilds the new file using the delta and then writes *exactly* the ranges that he received from the FileCache™.

Transfer locks to the EFS – When an application requests to lock a portion of a file, we may send the lock request all the way to the EFS. This may be done synchronously, meaning, for example, that only after the EFS granted the lock, the FileCache™ will grant the lock. In one embodiment, only after the lock was granted, the application can continue to write data to that portion in the file. Along with the lock request, the FileCache™ may also send read requests on that portion of the file (it could be more than one block). Along with the lock grant, the FilePort™ may send the

updated data for that block. In some embodiments, this may be used in order to maintain semantics – since a read that is done after a write (from any source) to the file should have the latest data of the file.

Transfer unlocks to the EFS – An unlock request that is sent to the FileCache™ is also forwarded to the EFS. Since the purpose of this request is to release other users that might be waiting to lock this portion of the file, fewer restrictions may apply. In order to optimize, this could be sent in an asynchronous manner.

2.3. Cache management

In some embodiments, a main consideration in caching appliances is that the cache size is significantly smaller than the real repository being accessed. Some embodiments may use, for example, a cache management algorithm which may utilize LRU (Last Recently Use) queue, where new coming data replaces the eldest stored one.

In some embodiments, a branch office might have different uses for a cache appliance, and thus different ways to handle the caches may be used. In some embodiments, if the usage pattern is defined, assumptions on the cache can be made. This may allow to further optimize cache usage.

In some embodiments, the following parameters can be defined per share:

- Cache priority [1..5] – files with higher priority (1 – highest) will be discarded from the cache only after files with lower priority (5 – lowest) were discarded.
- Change frequency [0-∞, 1/second] – Cache validation will only happen after the cache validity time (1/change frequency) has passed.
- Read only [False, True] – A read only parameter means that the share data can never be written to.
- Exclusive [False, <FileCache™ name>] – An exclusive share is a share that is being accessed only through a specific FileCache™ (branch office).
- Read-all [False, True] – All files in a read-all do not contain sensitive information and thus everybody can read their contents.

In some embodiments, the following implementation guidelines may stand for the aforementioned parameters:

- Cache priority – Some embodiments may evacuate space proportional to the priorities. For example, we may evacuate 3 times more space from priority 3 than from priority 1. Blocks with the same priority will be evicted according to the LRU (Least Recently Used data will be evicted first). This may prevent cases that files stay in the cache although they are not being used, and still maintains high priority data within the cache more than low priority data. For details about managing LRU prioritized storage, see Block Based LRU Caching below.
- Change frequency – The administrator may define the average change frequency she thinks is most relevant per share. If the share files are known to be changed once a day, one should define a change frequency of 24 hours. When a file is requested to be read, the cache is valid if the file was refreshed from the server less than its time-to-live ($TTL = 1/\text{frequency}$). If the file is requested for write access, then a lock request must be sent to the server, and thus we may also piggyback on its data validation. This way, a correct definition of a TTL may result in substantially optimal (or near-optimal) number of requests for data from the server.
- Read only – The administrator may define a certain share that no one is allowed to write to. This may apply only to users accessing files through the FileCache™ and not directly. However, when a user tries to access a file on a read-only (RO) share, he can only browse directories or open files for read. All other operations (create, move, delete, write, etc.) will result in an immediate "Access Denied" response, originating directly from the FileCache™, without going over the WAN. This optimization may speed up file open and access, along with ensuring that files and meta-data stay intact on that share, regardless of permissions.
- Exclusive – The defined FileCache™ is the only FileCache™ that is allowed to access files in this share. This may allow to make the following assumptions: 1) Files in the cache never expire (i.e. change frequency = 0); 2) There is no need to lock files at the FilePort™ and the EFS. Both of these optimizations may highly decrease response times to the user, since, for example, most of the transactions may include cache validation and file locking. Note that in some embodiments, there is no contradiction between a share being Exclusive and RO. The administrator should make sure that files in this share indeed do not change directly on the EFS.
- Read all – All files in a share with this property will be accessible by anyone, as long as this is possible. After a file was cached in the FileCache™, any user requesting the same file from the cache will be granted (for read) immediately, and without

checking the file's ACL. This may save the transaction and the check. Note that write operations, and other operations that must go through to the EFS, may not be approved by it, if not configured properly.

2.4. Save-as / speculative diff

Some embodiments may correlate different files that exist or existed at different times in the file system. When two files are correlated, if they have similar data, then sending a difference between them should (or may) suffice.

In some embodiments, the reasons that two files may correlate in terms of "close" data are twofold:

1. Applications, trying to ensure data integrity in case of a crash, use different files during a file save process;
2. Users tend to save different versions of files in different names – and all or multiple versions coexist in the file system.

By monitoring closely files deletion, creation and rename, some embodiments may find a heuristic that has good chances of determining that two files correlate. When the decision is made, a delta is calculated between the two files.

If eventually they do not correlate, then the delta calculation fails, and the system may fall back to sending a whole file. If the receiving entity doesn't have what it needs in the cache in order to build the new file, it re-requests the data, this time not allowing correlation of files. However, the last two scenarios are meant to be the rare case – just to make sure semantics is preserved. In some embodiments, in the common case, semantics will be preserved, while greatly minimizing the amount of data send over the WAN connection.

In some embodiments, speculative file correlation may be done, for example, in the following way:

When a file is deleted, its data is not dismissed, but saved in a special location for future potential correlation;

When a file is moved, its original name is saved for the sake of future potential correlation;

When a file is replaced (sometimes referred to as "truncate"), its original name and data are saved aside;

When data of a dirty block is sent to the FilePort™, an algorithm for evaluating correlation is activated; after the files are correlated, the FileCache™ calculates a delta between the two correlated blocks. If the delta is significantly smaller than the plain file, the delta is sent along with information about the block it correlates with.

In some embodiments, correlation may take into account one or more measures with different weights in order to consider candidates for correlation. The measures that "wins" most points may be the "winner" of this correlation. In some embodiments, if delta calculation proves that the files are not correlated, the system may try again with number two, three and so on in the correlation candidates list. However, in one embodiment, it may be preferred to make sure the algorithm finds the right file on the first try most of times rather than rely on trying again and again (for example, while the user is waiting).

In some embodiments, an algorithm to decide upon correlation candidates may maintain a limited size queue of filenames that were last opened on each session. Each file will get a score according to parameters, for example:

1. If it was more recently read than the others (for example, in a copy operation we usually read one file and write to the other).
2. If it was more recently written to than the others.
3. If it was more recently opened than others for the last time.
4. If it's still open.
5. If it was more recently closed than others.
6. If the candidate's name is similar to our committed filename (for example: whether or not its name is contained in our filename, as in the case of "Copy of a.doc", and if not - if there is a common substring starting either at the beginning or at the end of

the candidate, that is longer than a certain percentage of the shorter filename of the two).

7. In some embodiments, special treatment may be given to files whose names match known patterns, for example:

- i. If the file we're committing has the name ~WRD####.tmp or <8 hex-digits> – look for a *.doc file or *.xls file, respectively, that is still open on this session. Among such candidates – prefer the most recently opened or “dirty” file.
- ii. If we're committing a ~WRL####.tmp file (or, for example, an Excel equivalent), look for the most recently opened *.doc file.
- iii. If we're about to commit a file called “Copy of a.doc” or “Backup of a.wbk”, etc. – we may know exactly the filename we're looking for.
- iv. If it's a *.doc, *.xls, *.ppt, etc. file – look for files with the same extension, or the extension of the application's template file (e.g. *.dot).

2.5. Global Name Space

In some embodiments, users of an organization with multiple file servers (NFS only) in multiple locations usually need to know where their data resides. If all of the data resides in the data center, there are existing solutions for storage virtualization. If the data is distributed throughout the organizations, a WAN based solution is required. For this reason, we came up with the idea of providing a unique path for each file in the network, reachable from every location in the organization, by the same name – regardless of where it resides.

In some embodiments, each FilePort™ may maintain a map of file servers and shares. Each file server and share will have an additional entry by the name Global Path (GP). In some embodiments, there may be substantially no limitations on the GP; it need not be correlated with the file server and share. For instance, one embodiment may map EFS1:share1 to /dir3/share1, and also map EFS2:share3 to /dir3/share1/xx3.

In some embodiments, each FileCache™ has a list of FilePorts™ it contacts, and each FilePort™ publishes its own map of servers, shares and GP's. The FileCache™ combines the maps from all FilePorts, generating a single hierarchy of directories.

In some embodiments, each node in the hierarchy has one of three types: *real*, *pseudo* and *combined*.

In some embodiments, a *real* node represents a real share in an EFS file system. In the example above, /dir3/share1/xx3 is a *real* node.

In some embodiments, a *pseudo* node does not have any *real* files or directories in it. It is only there because it was mentioned in one of the maps as a "point in the way" in the path. In our example above, /dir3 is a *pseudo* node.

In some embodiments, a *combined* node has some *pseudo* and some *real* nodes in it. In our example above, /dir3/share1 is a *combined* node.

In some embodiments, the system prohibits the user from changing *pseudo* nodes by returning "Access Denied" response on any such a try.

In some embodiments, another use of this technique is data migration. The real location of the file can be quickly changed by changing the map. All the users will continue to work and see the same path as before; only now the file is at different physical location.

2.6. Installation & upgrade scheme

Some embodiments may use and/or implement various suitable installation and upgrades schemes

In some embodiments, the following parameters may be considered: the need to replace a failing appliance seamlessly, without losing data, the need to install more than one application on one appliance, the need to separate the application into different modules that may or may not coexist in one appliance – according to a given license.

Some embodiments may use and/or implement an installation and upgrade scheme that simplifies work both to the administrator, and to the customer. The main goals are: Separate application binaries and application data; Separate *roles* and *instances* (see below); Provide a modular scheme with maximum flexibility.

In some embodiments, when installing a module, it is installed in an *application directory* – this data is read only and is never changed. It is installed at <module_name>/<module_version>

under the common application binaries directory. This can be done separately from the license itself.

In some embodiments, when the user wants to *instantiate* the application, a script builds the following directory:

`<nodes>/<node#>/<module_name>`

Links are built for the binaries, and data directories are created. If data is important, this can be saved at a shared storage or other location external to the appliance itself. Each node receives a different IP, and acts as a different machine in the network.

This way, we may build multi-purpose machines (for instance, one machine can run FileCache™ version 4.5 with management version 1.2, FileCache™ version 5.0 with management version 1.3, FileCache™ version 5.1 with FilePort™ version 2.3 and management version 1.2).

In some embodiments, an upgrade may be done by first installing the new module version (does not interfere with normal users work), and then only changing the links under the requested node to point to the new version binaries).

In some embodiments, when an upgraded instance is created, an upgrade program runs. It analyzes the differences between the old version data files and the new version, and suggests the user one of the following: leave data intact if the data files format haven't changed; convert the data to the new format; erase the data files (in case an upgrade cannot be done to the data files). The user can choose what to do, or even restore the previous version in case he regrets.

2.7. Disconnected operation

For a cache based file system, there may be a need to provide methods to access files when the WAN connection is down. Some embodiments may provide read-only access to files that exist in the cache. In some embodiments, for example, the following details may be taken into consideration:

Disconnection event — A FileCache™-FilePort™ disconnection event occurred, for example, if one (or if at least one) of the following is true:

- Round-trip latency is higher than 1000ms for at least 5 seconds.

- IP layer reports more than 50% packet loss for at least 5 seconds.
- TCP layer cannot establish a socket.
- The FilePort™ does not respond to a simple ping (proprietary transaction, not ICMP) request.

A FileCache™-FilePort™-EFS disconnection event occurs if one (or if at least one) of the following is true:

- The EFS does not answer an ICMP ping for more than 5 seconds.
- IP layer reports more than 50% packet loss for at least 5 seconds.
- A file system protocol ping (if exists) is not answered by the EFS for more than 5 seconds.
- TCP layer errors prevent a file system protocol session, or a socket creation.

In some embodiments, detection of a disconnection event will occur immediately if an error is returned, and checked periodically every minute. It can also be manually set. When such an event occurs, the FileCache™ goes into a *disconnected operation mode*.

In some embodiments, during disconnected operation mode, the following changes in the system behavior:

- Cache is always valid, regardless of the time-to-live
- Request to open a file other than for READ access, will result in an "Access Denied" response.
- If the share is in read-all mode, access is always granted. Otherwise the op-cache (see below) will be checked. If the op-cache exists, it will be used, otherwise, the ACL (see below) cache will be checked. If the ACL cache does not exist, access is denied or granted, according to a configurable parameter.
- All CIFS sessions are closed upon a disconnection event. They can be re-created, but users need to be re-authenticated. If the FilePort™ is reachable, authentication is done as usual. If not, authentication may use the local authentication server, if exists, or a cached challenge-response sequence. New users may not be able to login, unless there is an accessible authentication server.
- All requests to change a file, data or meta-data, will be denied.

- Transactions that were in-transit during a disconnection event will behave as if the disconnection event happened *before* the transactions started.

In some embodiments, during disconnection operation, a test for re-connection will occur every 30 seconds. If all conditions for disconnection event are false, a reconnection event occurred. When a reconnection event occurred, all sessions are closed, and need to be re-opened by the clients.

In some embodiments, there is also a notification to users that the system switched to a disconnected-mode. This could be realized by either a message to the desktop (for example, using WinPopup protocol), or using a special client that is installed at the user's desktop.

2.8. Security

(i) User level Security

Some embodiments may include a WAN file system, proxy based, that authenticates users in path-through mode. When a client authenticates against the FileCache™ using challenge-response mechanism, its request for authentication is passed through the FilePort™ to the Enterprise File Server (EFS), which in turn returns a challenge.

The challenge is sent back through the FilePort™ and FileCache™ to the client. The client, believing that the challenge originated from the FileCache™, provides a response, which is transferred all the way to the EFS in a similar manner.

The EFS, believes that the response originated from the FilePort™, grants the authentication request (provided this was a legitimate request) and creates a session for the FilePort™, under the original user's privileges. The FileCache™ also does the same thing, and creates a CIFS session for the user.

This way, some embodiments may achieve a legitimate CIFS session that exists both between the user and the FileCache™, and between the FilePort™ and the EFS. These may actually be two different sessions, but they share the same privileges. In this way, substantially every operation that the user does on the FileCache™ can be reflected exactly on the FilePort™. All

authorization, auditing and quota management is done in the same way on the EFS as if the user was connected directly to it.

In some embodiments, the FileCache may or may not be a part of the Windows™ domain (or active directory).

In some embodiments, CIFS file servers may break a CIFS session with no locked files after a few minutes of inactivity. A client with locked files must send an echo message to the server, signaling that it is still alive. To preserve this mechanism, the FilePort™ sends echo requests to the EFS, as long as the FileCache™ sends I-am-alive transactions for this session.

In some embodiments, if the session breaks between the FilePort and EFS, upon next request to the EFS, the FilePort notifies the FileCache in the response that the session is not valid anymore. The FileCache in turn breaks the session with the client, forcing it to re-create it using the challenge-response mechanism. This is done transparently for the user, for example, by Windows OS. After re-initiating the session, Windows clients repeat on the original request.

In some embodiments, if the session breaks between the user and the FileCache, the FileCache stops sending I-am-alive transactions to the FilePort on that session. The FilePort will not send echo messages on this session anymore, and the EFS will initiate a session close after the timeout (usually between 4 and 15 minutes, configurable for Windows™ servers).

In some embodiments, for other authentication mechanisms that do not use challenge-response methodologies, other methods apply. For example, for Kerberos, one need to configure the system to work with *forwardable tickets*, so the tickets can be forwarded from the FileCache™ to the FilePort™ to the EFS.

(ii) Branch Level Security

In some embodiments, in addition to working in path-through mode, there is another mode of operation for a caching system.

Some embodiments may have a separate special user per each installed branch. The user will have a superset of credentials that exist in the branch. The FileCache™, upon connection to the FilePort™, will identify using this user. The FilePort™ will validate the user using the

authentication server, and will connect to the EFS using that user. All operations done on files will be done on behalf of that user.

Some embodiments may take into account, for example, the following:

- i) User quota (if being used) is not preserved. Since we work on files using different user, in some embodiments there is no knowing of the originating user, and we may not manage his quota changes.
- ii) File owner is not preserved. When new files are created, they are owned by the special branch user. In order to avoid accessibility problems, the FileCache™ adds the original user as "Author" of each file created.
- iii) Branch security always preserved. The special branch user privileges define a limit on what a branch user can do on files. If a privileged user goes to the branch, he is still limited by the special branch user's privileges.
- iv) Session break handling. If a session breaks, all the files are closed and locks released. In case of a sporadic WAN connection, this can happen relatively a lot. Using branch level security, the system knows how to re-create the session if the connection is re-gained, without the client's intervention. Moreover, if files were locked by the session, the locks are re-created (unless the files were changed in the middle)

2.9. Quota support on FileCache™

In some embodiments, FilePort™ synchronously updates EFS with Write transactions it gets. Therefore, being pass-through authenticated, FilePort™ supports user's quota naturally. On FileCache™ side, however, write requests are not always (for Short Term File Handling, or stfs – just never) immediately verified. In order to avoid quota limits violation, FileCache™ manages these limits by itself.

In some embodiments, it handles a list of <user, share> entries; each entry holds an actual quota limits which is updated periodically from the FilePort™. In addition, the entry is updated during the operations that affect the amount of share free space (namely: write, set file size and delete).

In some embodiments, in Win32 semantics, the user that is charged for the quota is file's owner and not necessarily the user that performed the actual change. Therefore, In some embodiments, FileCache™ uses the file's security descriptor in order to update its quota list.

2.10. Non standard CIFS server support

Some CIFS servers do not provide some of the CIFS functionality. There are also certain parameters an administrator may set on a standard CIFS server that will change its behavior. When working using windows clients on a local network, the difference in behavior may go unnoticed. But when working through a cache, some problems that need special treatment arise. Therefore, some embodiments may take into consideration, for example, the following:

Unsupported GetFileSecurity() – CIFS servers like Novell NFA, Samba on certain UNIX systems, may not maintain full Windows-NT style ACLs. Therefore, in some embodiments, a cache system may not possibly know in some cases if a user can read a file that exists in the cache. In order to support such a system, some embodiments of the invention may use and/or implement the following functionalities:

- i) *Path-through authentication* – Using this mode of authentication, the FilePort™ fools the EFS to believe that the real user is the one doing the operations on the files. Therefore, the EFS will grant an operation on a file if-and-only-if the user can do the operation and vice versa (the EFS will deny). In one embodiment, using another method of authentication may make it relatively difficult to decide if a user can grant an operation or not.
- ii) *Operations cache* – For every operation a user requests we cache the following quadruple – <user, file, operation, result>. Whenever an operation is requested again, if the first three parameters match one of the cached quadruples, we use the cached result instead of turning to the server for advice. In order not to create a security breach, a change in the file data will cause an invalidation of the operations cache.

Personalized browsing – Some CIFS servers (Novell NFA) have a feature that each user can only see the directories he has access to (browse access at least). In some embodiments, this may a difficulty for caches that should provide services to more than one user (since information is per-user). In order to show the right information to the user, in some embodiments, several solutions may be used, for example:

1) Use a special user for directory browsing at the FilePort™. This user will be defined in the FilePort™ and will have browse privileges on all directories. All users will see all directories (this is a change in behavior, but may be minor), but, as before, will only be able to browse the directories they can access to;

2) Have a different cache per each user, for files meta-data. Although this will solve the problem with no behavioral changes, it may result in a relatively less efficient caching.

2.11. House keeping

Some Windows™ servers may use a common behavior of saving a file to persistent storage 4 seconds after the last write to the file. The definition of file system semantics usually does not require such a behavior from the server. However, some applications rely on this behavior, and do not flush() file data when they want the file to be saved (for example, MS Project 2000 does not request to flush the data of a file that is still open).

In order to support these applications, some embodiments may run a special background thread (named *housekeeping thread*), that searches in the list of open files, if there are outstanding write requests that were not sent to the server. If there are such requests, an artificial flush() is generated, and the data is sent to the server. This way we achieve both similar behavior to Windows™ file server, and do not hold the user waiting for such requests.

3. Optimizations

3.1. Read-ahead and write-back predictions

In some embodiments, the WAN file system of the invention may utilize a set of optimizations that may be based on usage patterns of the common Windows and Office applications.

For example, when Windows Explorer opens a directory, it fetches all the files in it. We know that it is needed to display a file-associated data (preview, icon, etc) and we know which areas are read in which kinds of files. We also know that some applications (for example, Word, PowerPoint, MP3 players) may allow users to start working before the entire file has been read. Consider a large or non-cached directory or file; some embodiments can significantly improve user experience by supporting predictive transport of a data needed.

In some embodiments, FileCache™ utilizes the latency it is going to waste in each transaction and always tries to attach additional requests, based on its own prediction decisions. For instance, it makes sense to request some blocks and file's metadata along with an "open" transaction, or parent directory's metadata and free disk space during the "delete" transaction. It also may be wise to get an actual status of a neighbor blocks during block-related transactions or just get another file's information when it looks like an Explorer browsing pattern.

In some embodiments, on the other hand, FileCache™ may be always aware of a CIFS timeout problem (see above) and thus avoids of collecting the data it will need to commit. When this data overpasses the certain limit (calculated on-demand due to the current network and file conditions), the data is committed on the FilePort™.

We discovered that some Windows Clients tend to ignore the "Close" results; and in real life this fact may not interfere in some cases with file-system and application semantics. So, in some embodiments, FileCache™ never blocks on close requests and attaches them with the next coming transactions. Of course, when the FileCache™ gets "Open" and it still has such a "Close" pending from the previous request, it may extinguishes both. Taking into account that some windows applications use to open/close the same file a numerous number of times in a sequence, this approach of some embodiments of the invention may be extremely useful.

3.2. Short-term files handling

In some embodiments, some applications often hold their intermediate data in temporary files or so-called *stf* (short-term files). Because these files are accessed rapidly and are heavily used, but on the other hand, they are normally deleted when the application finishes its work, the system in some embodiments may hold them locally on the FileCache™. When the file is created via the FileCache™, the last may decide to create the file as *stf*. (In some embodiments, this decision is based on file's name)

In accordance with embodiments of the invention, there may be at least two considerations with short-term files:

The first consideration refers to parent directory management: directories that FilePort™ sends to the FileCache™ do not include the *stfs*. Therefore, for each directory that contains *stfs*, the FileCache™ manages separate faked directory and merges it with the real directory during

directory read. When looking the file up, the FileCache™ always searches in real directory first and then in the *stf* directory.

The second consideration refers to the fact that certain kinds of applications tend to rename *stfs* to the regular files (for instance MS Word save scenario: "open 1.doc file; copy it to 1.tmp; delete 1.doc; rename 1.tmp to 1.doc"). In that case all the data that was stored locally has to be transferred to the FilePort™ - and at once. Of course, if the file is big enough to cause a CIFS timeout, the application will break; and, in some cases, write-back can not be applied here. Instead, in some embodiments, the system chooses not to define such temporary files as *stfs*, and a file that has been created as *stf* will remain *stf* forever.

3.3. NFS handles

In some embodiments, a file server (usually NFS server, but may be others too) may need to supply unique handles for its files. For every file accessed by the client, the client receives from the server a unique ID. The client then uses that ID to access the file. Some NFS servers do not require an `open()` transaction before read or write operations and thus the unique ID is used. This means that a NFS file server must be able to find the file data upon a request that contains only its ID. Most NFS servers use the real file system for that matter, i.e. they give out the actual block number (*inode*) to the client.

In some embodiments, a caching file system that supports NFS may not do the same, since it is only caching and does not store the files physically.

In one embodiment, a solution might be to use a database that relates all the files and their unique ID, but this approach may have a few drawbacks:

1. It may be relatively slow to store and/or retrieve such amounts of information;
2. There is no easy way to identify files that have been moved - we would like to use the same unique ID in order to preserve regular NFS server behavior;
3. It is hard and time consuming to know which entries we can evacuate from the list. Since this is a caching device, every data stored should be cached out at some point in time.

Another option is to use the same unique ID that comes from the server, but this may have a drawback - different servers might use the same ID (since the ID is unique per server).

Due to some limitations of approaches above, some embodiments may use a shadow directory. Since we have a unique ID for every server (server-ID) and a unique ID per file in every server (file-ID), we create a special file named "<server-ID>-<file-ID>". The underlying file system gives a unique ID per every file (*inode*) since it is a regular storage system. Some embodiments may use the unique ID of the shadow file, that gives us a unique, consistent, persistent ID for every file that is accessible through the cache. Trusting the underlying file system (could be any of ext2, ext3, jfs, xfs, reiserfs, reiserfs4, and in one embodiment ext3) is a simple and easy solution – since it does that anyway, and is optimized for this kind of things.

3.4. Security descriptors hash

In some embodiments, in addition to caching files and files structure (meta-data), the system caches *security descriptors* (SD). An SD contains information about who is entitled to do what operations to that file.

In some embodiments, caching SDs may allow to: 1) be able to analyze the SD and decide if a certain user can do a certain operation on the file; 2) send the SD to the client when it issues a `GetFileSecurity()` request; 3) Provide information about the file's owner, in order to support quota.

We have discovered that in some cases, even for large deployments with many files, there are very few different SDs. In order to save in space and in transactions we save the SD in a special directory, under a file by a name identical to the SD hash (hash can be computed by any suitable hash algorithm, such as the standard MD5 hashing algorithm).

In some embodiments, in the file structure, we have a field that contains the SD hash. When a new file is read, its SD hash is computed by the FilePort™ and sent back to the FileCache™. If the FileCache™ already has this SD in its cache, it doesn't need the FilePort™ to send it over. Since the ratio between different SDs and different files is very close to 0, we save a lot of transactions and bandwidth by caching each different SD only once.

In some embodiments, we do not need to maintain reference count of any kind on the SDs. We save them as part of the LRU'ed cache – which assures that unused SDs get evicted from the cache eventually.

3.5. Directory lookup cache

In some embodiments, a client issues many requests for file lookups. This may actually be the most used request from a client. Many applications even search for files just to make sure they do not exist.

In some embodiments, optimizations may be used here for performance reasons. In order to do so, we use *positive caching* and *negative caching*.

In some embodiments, positive caching means that we save for every successful request the fact that the specific file was found in the specific directory, and the result of the search (the file unique ID). When another request to search for the same file arrives, we search in this cache (we named it *directory entry cache*) if this file was already found, and if so, return the previous result.

In some embodiments, negative caching means that we save for every failed lookup request the fact that a certain file was not found in a certain directory. When subsequent request to lookup for the same file arrives, we search in the cache, and if it is found, we return immediately the result (that the file does not exist). Consideration needs to be taken in invalidating this cache. For example, when a directory is changed (we know this, for example, according to its version number), all the positive and negative caching for this directory become invalid. Another option is to go over all the caching for that directory and update it, but in one embodiment we may delete it, since it is faster and does not incur any additional delays.

3.6. NFS open/close optimizations

NFS version 2 does not support open/close transactions. Since the Unix file system (and windows) do require an open transaction before read/write requests, and close when the data is flushed, NFS clients tend to open the file before every read/write request, and close it immediately afterwards.

When the storage is local to the server, this goes unnoticed, but on a WAN file system this delays work a lot if implemented naïvely.

In some embodiments, when using the system to serve NFS requests, we may ignore close requests (and subsequent open requests), and use a different thread to perform them. Since an

NFS client may choose to execute many subsequent read requests, we may save many adjacent close-open transactions.

In some embodiments, when a close request that originates from a NFS server arrives, we close the local (FileCache™) file handle, and do not send anything to the server. If after a few seconds (for example, 5 seconds) an open request arrives, with the same attributes as the previous open, we may re-open the file and notify nothing to the FilePort™. In some embodiments, if no open request arrives within those 5 seconds, the special thread (named *housekeeping thread*) sends the close request to the server.

In some embodiments, using this approach we may gain a lot of performance, since we save at least two transactions for every additional read/write subsequent request from the client. On the other hand, in some embodiments, no semantics problems arise, since there is no open/close logic exposed to the client anyway, so there are no requirements on the server on when to save the data to persistent storage. In some embodiments, one exception is a flush() request, that we should honor synchronously.

3.7. Dynamic compression and diff filters

In some embodiments, each file that is sent to the server goes through, for example, two compression functions: one that tries to compare it to another file, and send only the delta between them (*diff*); another that simply compresses (using the known Lempel-Ziv algorithm, and the known zlib implementation) the file.

In some embodiments, we may apply both of the methods – regardless of which one has failed (failure means that the total save in file size was not worth the time and effort i.e. CPU cycles invested in the compression).

In one embodiments, in which there is no easy way to anticipate the outcome of a compress or a diff activation, we may try to save unnecessary activations of the algorithm.

For this we may use a dynamic filters system. For example, whenever the system runs an algorithm on a file, it saves the number of compressed (or delta) bytes divided to the original file size, and the file extension (i.e. the string after the last period (.) character in the file). During its

work, the system gathers information (average compression ratio) about the compressibility of certain types of files.

In some embodiments, if files have compressibility lower than a certain threshold (for example, 70% for compression and 20% for diff), the system does not run the algorithm next time it finds such a file.

One embodiment may also set a static set of rules that will work very well, without the hassle of a dynamic system. An example for a set of extensions that need not be compressed is *.avi, *.zip, *.mp3, *.ogg, *.mpg.

In some embodiments, in order to be able to change decisions, the system may slowly increase (artificially) the compression ratio for each type of file it chose not to compress, until it passes the ratio again – and another test is made. The results are saved on a persistent cache, so the system gets to be optimized after a few days of use to the types of files the customer actually have.

3.8. Mirroring

A cache based file system may have means to pre-populate the caches. Pre-populating the cache will give higher cache-hit ratio and better performance for the users.

Some embodiments may populate the cache by running a program that scans the relevant directory tree, and reads all the relevant files there. Traversing the directory tree this way will result in the cache being populated at the end of the traversal. If this program runs at night times, the users come in the morning for a fresh cache.

However, with this approach, every file is read separately, and it requires a special transaction; thus for n files in the system, we will have around kn (k should be a small one digit number depending on the implementation) transactions to do so.

In some embodiments, another approach may be used, for example, *Mirroring* mechanism. This includes a special transaction that is capable of synchronizing the contents of many files. When the FileCache™ wants to update its cache, it runs the mirror transaction that includes information about all the files that need refresh, along with their cached version numbers. The FilePort™ responds with a list of updates – could be any of "No update, you have the most recent version"

or "You have an old version, here is a delta to patch for the latest version". The amount of files to be sent per transaction can be configured. One embodiment may update 100 files each transaction.

In some embodiments, FileCache™ needs to follow closely upon directory updates – if files were added to the directory, they need to be added to the next round of mirroring.

In some embodiments, another optimization is to find out, according to the directory information, which files did not change at all, and therefore do not need an update.

In some embodiments, a different way to implement such a mechanism may be to aggregate a set of requests to one transaction. There will be many READ (or OPEN) subsequent requests that will be sent in one transaction and the FilePort™ will respond as usual to all the requests (in one response transaction). In some embodiments, this implementation only requires the implementation of multiple requests mechanism, but does not require a special MIRROR transaction.

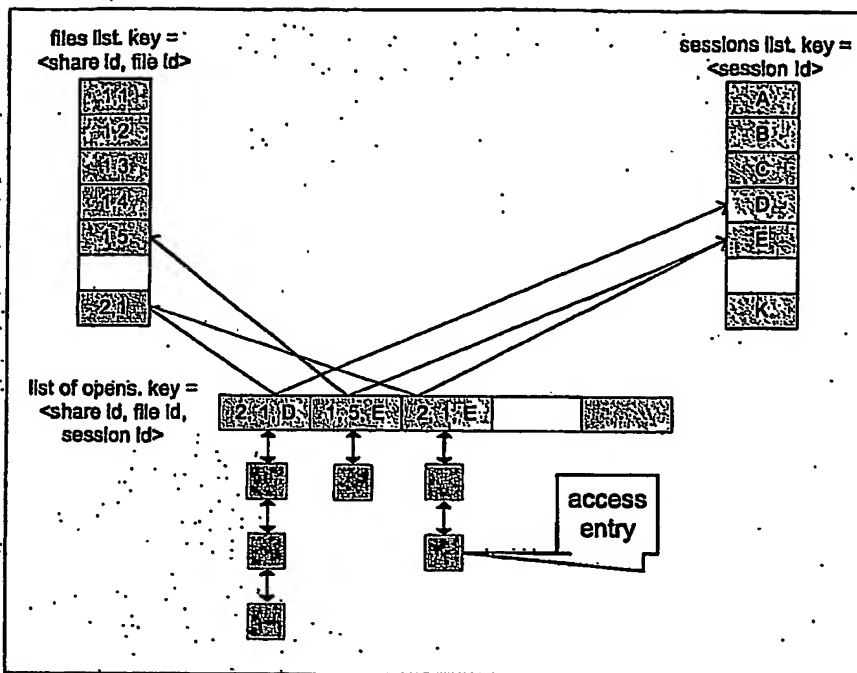
3.9. Open handles multiplexing

In CIFS the files may be open with any combination of Read/Write/Delete access and Read/Write/Delete sharing. In some embodiments, in order to optimize the traffic, we may use a simple mechanism of access/sharing arithmetic (assuming that initially all files have an access for none and sharing for all), for example:

When the file is opened FileCache™ looks in its handles list and checks if no sharing violation has been occurred; then it issues a new handle. Only in a case, a total file's access or sharing mode for a certain session is going to be changed as a result of open, FileCache™ decides to notify a FilePort™ with Open transaction.

When the file is closed FileCache™ removes it from its handles list and destroys a handle. Only in a case, a total file's access or sharing mode for a certain session is going to be changed as a result of close, FileCache™ decides to notify a FilePort™ with Close transaction.

The handles list (mentioned here above) has, for example, the following structure and rules:



In some embodiments, File, Session and Open lists are based on hash tables with appropriate unique key. The sizes of these tables are fixed and are derived from the site license.

In some embodiments, each time the file is opened, an access entry is allocated and a handle to this entry is issued for the caller. In addition, for the first open of this file the list generates a file entry and for the first open by this session the list generates a session entry. The first open of the file by the same session also generates an open entry. And vice versa: the last close of the file causes the list to remove the file entry; the last close by the session causes removing appropriate session entry. The open entry is removed too, if no more access entries for the same <share id, file id, session id> persist. Access entry is freed on close, and the handle to it becomes invalid.

In some embodiments, in order to avoid iterations while total access and sharing counting, file and open entries hold the counters of access and sharing instead of total masks; these counters may be easily converted back to masks.

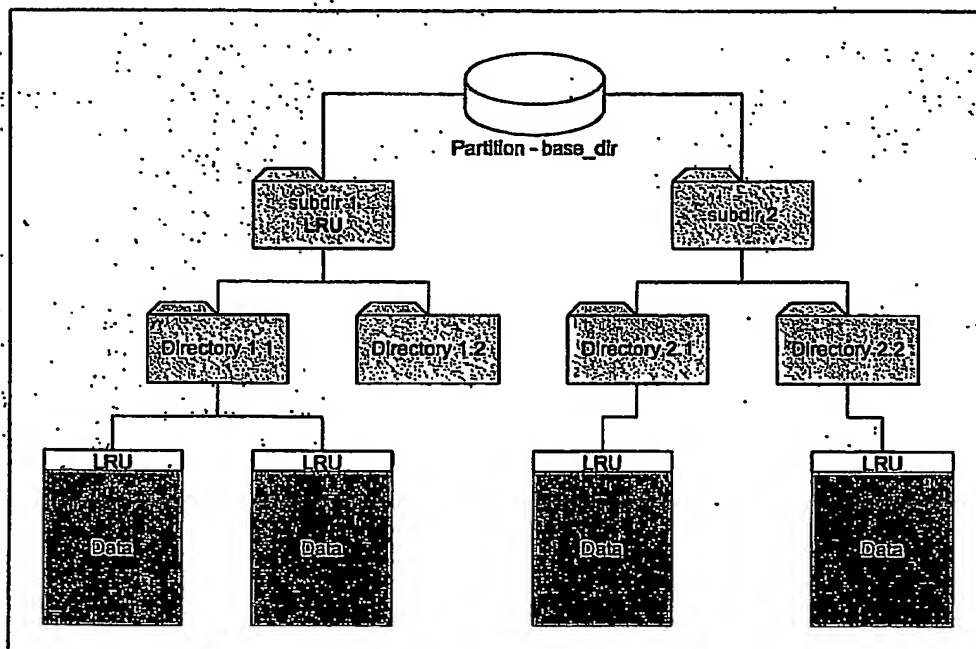
In some embodiments, in addition, the list provides ability to scan all access entries that belong to the same session and to perform a specified operation on this entry. (For example, remove it, or

update its data). This may be especially useful when FILECACHE™ and FILEPORT™ discover that the session has changed its state from valid to invalid or vice versa.

4. Low level implementations

4.1. Block based LRU caching

In some embodiments, FileCache™ and FilePort™ may share the same (or similar) mechanism to cache blocks of files, while maintaining performance requirements. In some embodiments, blocks are stored on disk – each block in a separate physical file, named by the key that defines this block. There are separate directories for each type of the blocks: *plain*, *dirty*, *diff*, etc. Directories may have various attributes, such as *LRU*, *to-be-deleted-on-reboot*, *permanent*, and are unified into partitions. For example, in some embodiments, the illustration below may apply:



1. Path construction

In some embodiments, in order to ensure good dispersion of files within subdirectories (big amount of files in the same directory slows the work on certain file systems) the files are situated

within the tree of subdirectories (see Directory 1.1 on the figure above). All files reside under the partition's base_dir, in their subdir.

Under that subdir the path construction is as follows: we take the given key, break it down to 2-characters strings (the last one may be shorter), and insert a '/' in between, so that all these 2-byte strings but the last one are directory names. Therefore, the key must be an alpha-numeric string.

2. Access to stored data

In some embodiments, in order to achieve flexibility, the cache subsystem is completely agnostic to the data it stores, it just enables access to the file from the point where the LRU section ends, so that if LRU gets x bytes, each read write request will be performed with $\text{shift} = x$. Since the system shares disk resources for all kinds of cache, and, therefore, use a single storage instance, all cache types share the same key between them.

3. Boot up recovery

In some embodiments, since the LRU lists themselves are maintained in the files rather than in memory, the storage module maintains a recovery file during each LRU operation. This recovery file is read at initialization time and acted upon, to ensure that if we crash in the middle of an LRU operation, after reboot we know how to fix the (possibly) broken LRU and get back to a consistent state (either to the state before the operation we were doing when we crashed or to the state after it).

4. LRU

In some embodiments, files are discarded not only according to their LRU status, but also according to their share priority (see Cache management), for example, in the following way:

We keep priority meta-nodes in the cache LRU queue; one meta-node per priority. Let's mark them M_1, \dots, M_n ($n=5$). We maintain pointers to these meta-nodes at all times. When the cache is empty, we start with a queue that looks like this:

Head $\rightarrow M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_n \rightarrow \text{Tail}$

The cache operations are performed as follows:

Insert: Calculate entry's priority according to its share priority, type, state and data size. If (i) is its priority, insert it right after (to the right of) M_i .

Touch (any use of the file that makes it the most recently used one): if the file's share priority is (i), move it to be right after (to the right of) M_i .

Delete: just delete the file out of the queue.

Discard (to free space): Start discarding from the *Tail* side, and discard as many files as needed, till their accumulated sizes pass the required space to be cleared. For each file we discard, if its priority is (i), move the first k regular nodes from the left of M_i to its right, where k is a fixed constant number. "Pinned" files are situated between Head and M_i . The LRU never removes the files that are situated left to M_i .

Note that in some embodiments, the discard operation makes the higher priority files drift down the queue, passing the meta-nodes of lower priorities. Thus, with time, the files along the queue will be of mixed priorities. The higher priority files just get a better "head start" when they are inserted or touched, so that they have a longer way to drift with LRU before they get discarded.

In one embodiment, consideration may be given to the starting period, when the cache has just filled up for the first time, before enough discard operations have been done. During this period of time, the queue is still pretty much sorted by priorities, and the first files to be discarded will be the lower priority ones.

In some embodiments, to get this phase through as quickly as we can, we can set k to a value higher than 1, say, 10.

4.2. Delta (diff) calculation between blocks

As mentioned above, in some embodiments, we substantially never (or most of the times) compare files but rather compare appropriate file blocks.

In some embodiments, there may be, for example, two basic functions in the scope of diff calculation: the first gets two blocks *block1* and *block2* and returns *diff=block2-block1*; the second gets *diff* and *block1* and returns *block2*.

The binary diff is essentially the $O(n^2)$ issue, yet our approach gives a $O(n)$ complexity.

1. In some embodiments, the diff is actually a stream of tokens, of two kinds:

reference tokens: include an index into *block1*, and the length of the referenced string. When patching the diff on *block1* in order to reconstruct *block2*, we copy this string from *block1*.

explicit string token: include a string that appears in *block2*, and which could not be found in *block1*.

2. In some embodiments, the diff algorithm uses the following intermediate data:

hash table: an array of about 64K entries in size, each entry consists of an index into *block1*, and the entry's index is HASH of the 8B-word at that index in *block1*.

token buffer, es buffer: memory buffers used to store token and explicit string data, before they are compressed to create the final *diff*.

3. In some embodiments, the diff algorithm is a three phase algorithm:

Phase 1: Create a hash table of entries within *block1*, so we can access strings in *block1* directly (in $O(1)$) without searching for them in the block (which would be $O(n)$). The chances of finding the string we are searching for, assuming it exists, are related to characteristics of the hash table.

We hash, for example, 8B-blocks of *block1*. One reason is that this is the minimal size in which there is enough differentiation between blocks. Experiments with, E.g. MS-Word files showed that 4B aren't enough (because they represent only two Unicode characters). Larger blocks will consume much more CPU to hash.

In one embodiment, our benchmarks show that the hashing takes a considerable percent of the total diff time. In order to reduce the hash time, we actually only hash 1/19 of the overlapping 8B

words in *block1* (for example, in a 1MB *block1*, there are 1MB-7 overlapping 8B-words. In one embodiment, we will only hash about 53K of these words).

The index distance between two consecutive hashed words was chosen to be 19, which gives good results, since each offset within an 8B-word (i.e. offset 0,1,...,7) is reached once in 8 iterations (because $\text{GCD}(8,19)=1$), but also because the sequence of offsets spans the 8B-word well (i.e. the first offset is $19\%8 = 3$, which is in the first half 4B, the second is $38\%8 = 6$ which is in the second etc).

In one embodiment, we may rely on the "backwards comparing" technique in Phase 2, to overcome the effect of hash misses that result of this very partial hashing. One reason we may hash blocks in all offset into the 8B word, and don't just hash blocks on word boundaries, is that in phase 2 we advance by 4B-words at a time, but we still may wish to detect, for example, blocks that have their index shifted by one byte between *block1* and *block2*.

In some embodiments, we traverse *block1* backwards, because we want the easiest (smallest index) appearance of a 8B-word in the block to be the one that's in the hash table, and for performance reasons, we want to avoid checking that the hash entry is "empty". One reason we want the earliest appearance of a word to appear in the hash table, is to detect "runs". A Run is a long string of identical bytes, typically of '0' (often also 'FF'). This way, one of first words of the run will be cached, and there is a good chance we will be able to detect the whole run in Phase 2.

One hash function which may be used is (mod FFF1). FFF1 is a prime number. Z-FFF1 is cyclic group, ensuring that the hash is evenly distributed (that is, without a-priori knowledge of the data distribution in *block1*). As the calculation of the hash function may consume a major part of the CPU time for the diffing, we may make sure it is carefully implemented - reverting to coding in assembler...

In some embodiments, for example, for performance reasons, we don't initialize the hash table - so at the end of the hashing function, entries either contain an index into *block1*, or contain junk. We leave it up to Phase 2, to determine if an entry is valid or not.

Phase 2: In some embodiments, in this phase, we traverse *block2* from beginning to end, and try to find strings that are identical to strings found in *block1*, albeit no necessarily at the same index.

For each such string found, we output (to the diff) a token called a *reference token* that indicates the index and length of that string in *block1*. If no such string is found, we simply output the *block2* word as an *explicit string*. Several consecutive *block2* words maybe grouped into an *explicit string token*.

We loop through *block2*, and for the current 8B-word (called *datum*), we find the longest string in *block1* at the index `hash_table(HASH(datum))` that is identical. It may be the case that this entry of the hash table contains junk, or that it contains an index into *block1* that contains a word other than *datum* (because two different *datums* hashed into the same hash table slot), in which case we output an explicit string to the *es buffer*.

In some embodiments, up to 128 consecutive explicit string 4B-words are described by one *es token*, which is output to the *token buffer*.

In some embodiments, if an identical string of some length is indeed found in *block1*, then we output an *Reference Token* to the token buffer. In some embodiments, we also check *backwards*, to see if the string found actually starts earlier than we found, in which case we may delete previous tokens written to the *token buffer*, and potentially previous explicit strings written to the *es buffer*, and replace them by one large *reference token*.

In some embodiments, we may have only two kinds of tokens and may not attempt to have different kinds of *reference tokens* with different lengths. In some embodiments, we leave it up to phase 3 compression to compress the *token buffer*. However, in one embodiment, we do organize the bytes within the *reference token*, to help an entropy compression algorithm to compress better.

Phase 3: In this phase, we compress the *es* and *token buffers*, and add a header, creating the final diff data.

For example, compression can be done using any suitable compression algorithm, such as zlib (Lempel-Ziv algorithm) using the maximum speed (9) since it changes only slightly the compression ratio, while saving on 30% CPU.

In some embodiments, we compress the *token buffer* and the *es buffer* separately - that gives, for example, a total compressed buffer size which is about 10%-15% smaller, because of the different characteristics of these two buffers.

In some embodiments, another improvement is to supply the diff algorithm a list of ranges in the file that were changed. The diff can then run only on those areas, and not waste time on areas in the file that were not changed.

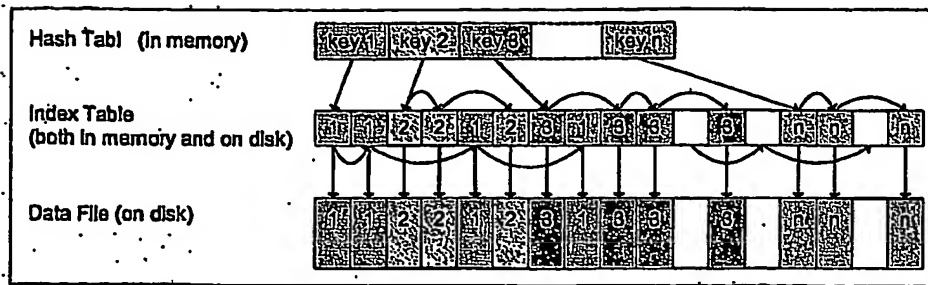
In some embodiments, dividing the file into blocks simplifies a diff procedure in case that some data was replaced in the file - then only changed blocks will be diffed. For example, in the worst case, when the data was inserted or removed in block k (in file of n blocks), all the blocks from k and further will have a diff. In order to overcome this problem the diff may be provided with different dictionaries (in this case $[k-n]$, or the entire file).

4.3. Disk Hash

Some embodiments may include a low level data structure for persistent storage that may include the following features:

- Consumes a (relatively) very low amount of RAM per each object in the data structure. Only some of objects have their data loaded into RAM, and the rest have their data residing on disk. However, RAM consumption is still $O(n)$.
- Do substantially all operations on objects in $O(1)$ time. In one embodiment, this may not always be accurate, for example, due to hard drive seek time, which may result in a relatively poor $O(n)$ time complexity in some cases. However, in some embodiments, if we get a high enough hit rate on the objects in RAM, access to disk can be largely avoided.
- Allow multiple search keys, and do not limit modification of an object's in any way, including, for example, changing its key fields.
- Have a double LRU discarding mechanism - both from the RAM cache, and from the entire list itself. In some embodiments, this mechanism is activated either when the list fills up, or when one of the metrics exceeds its limits.

Some embodiments maintain at least a three-tier data structure, such as:



- A Hash Table - of length of a certain fraction of the maximum number of objects. In some embodiments, for each key, we use HASH1(key fields in object) to get to a hash entry. This entry, for example, may include (per key) an index into the index table (see below), that consists of the beginning of a double-linked list of all objects that relate to the same HASH1, under this key. This table resides both in memory and on disk. It is loaded into memory at initialization, and is updated on disk each time it is modified.
- An Index Table - of length of exactly the maximum number of objects. Each entry in this table relates to an object's data on the data file that resides at the same index. Occupied ("used") entries in this table may be connected, for example, via a double-linked list, implementing an LRU mechanism for discarding objects. In some embodiments, objects are discarded to make room for a new object, either when this table is full, or when some metric is exceeded. Unoccupied ("unused") entries are connected via a linked list of "unused entries". In some embodiments, each entry also contains the index of the object in memory, if such object is loaded in memory (this isn't always the case). In one embodiment, the entry at index #0 is called the superblock, and its fields are used for administrative purposes - e.g. being the beginning and end of the various linked lists. Each entry, for example, may also include (per key) the HASH2(key fields in object). This is to increase the probability during a search-by-key, that this entry (reached through HASH1), indeed contains the correct key. If both HASH1 and HASH2 are correct, in one embodiment, we may load the object from disk and compare the key fields themselves. -Hence, in some embodiments, we may have 100% assurance that when searching an object with a certain key -

the object returned indeed has that key. This table may reside both in memory and on disk. It is loaded into memory at initialization, and is updated on disk each time it is modified.

- A Data File - containing the data (and a small header) of all objects of class. In some embodiments, the data file is updated substantially every time an object is checked in, and object data is read from the file when a search for an object is conducted, and that object does not currently reside in memory.
- In addition, in some embodiments, we create an associated Queue search index - to implement the LRU mechanism for the objects in memory.

5. Applications

5.1. Backup consolidation

Some organizations have and will continue having remote file servers at the branches. Backing up these systems, sending the tapes to an off-site location and/or restoring data when needed is cumbersome, cost-ineffective operation.

Using the *Backup Consolidation* product in accordance with some embodiments of the invention, one can back up the remote file servers in the same manner he backs up his data center.

In some embodiments, installation is done by installing the FilePort™ at the branch office and the FileCache™ at the center. The FilePort™ is configured to give access to the same share that needs to be backed up. The FileCache™ at the center is configured to connect to all the remote FilePorts™. The administrator configures his centric backup software to back up the shares that reside at the FileCache™. The shares are configured as read-all, non-exclusive, read-only (unless a restore function is also needed through this method).

In some embodiments, when the backup software tries to read the files from the FileCache™, the FileCache™ make sure that the files read are the latest files exist at the remote branch.

Using the cache and other suitable optimizations, we make sure that bandwidth usage is optimized over WAN, and only the data that was really changed since the last run is indeed transferred over the WAN.

5.2. Old versions retrieval

In some embodiments, the system can be used in order to retrieve old versions of files that were saved through the system. This provides, for example, the benefits of automatic version management for users, without involving the administrator.

One advantage of embodiments of the invention over standard backup solutions, or even standard snapshot solutions, is that it is event-driven and not time-driven. A regular backup or snapshot solution is configured to happen every x minutes. If you happen to need a file that was saved and deleted within less than x minutes, your file will not appear in the backup listing. Our solution in accordance with some embodiments of the invention saves *every* version of the document that existed.

In some embodiments, implementation may include, for example, the following: every directory will contain an additional *pseudo* directory named "archive" (or any other name). The directory will be added by the FilePort™.

When the user tries to open the "archive" directory, its contents is dynamically built the following way: The FilePort™ reads the file listing of the same directory "archive" is in, and prepares a list of all the documents that have different versions in its cache. In some embodiments, since the FilePort™ saves all the delta's calculated, and the time of the calculation, such a list can be relatively easily built from the cache.

For each such file, the FilePort™ creates a pseudo-directory, by the same name as the file. When the user browses into that directory, it sees a list of pseudo-files, but their names are dates and times, that represents the times that the file was saved.

Opening these files (for read-only) will get the user with the version as existed at that time.

For instance, if we have a directory structure with \documents\A.doc and documents\B.doc, we may also see \documents\archive. Entering the latter directory, we may be able to see two additional directories: \documents\archive\A.doc\ and \documents\archive\B.doc\ and inside A.doc we may be able to see the following files: [July 27, 2002, 5:20pm].doc [August 14, 2002, 11:18am].doc [Sep 22, 2003, 8:19pm].doc. The modification times of the files will be the same as the file names, to ease sorting.

In some embodiments, when the user tries to open a file, the FilePort™ sends only what the FileCache™ needs to build the file up to the version number requested. In order to do so, it uses the cached version number of the FileCache™ in prepares an appropriate "delta" in order to get to the requested version. Note that in one embodiment, the delta might *reduce* the version number that the FileCache™ has in cache. The FileCache™ will use the cache it has for the original file.

5.3. Virtual remote client

In some embodiments, another use for such a system would be to install it on a personal mobile computer, and users can use it at home (as opposed to an appliance, or a dedicated computer), in addition to doing their normal work. In accordance with some embodiments, several implementations are possible:

A GUI application – Some embodiments may allow to write a GUI application that knows to read and write files, and browse and change directories (similar to Windows™ explorer), and activate the FileCache™ engine upon such requests (similar to the way it is activated from the CIFS or NFS server on the appliance). It is noted that this is not a file system, but a new application that may require special training by a user.

A user-mode file system – Several methods may be used to write a file system while still using user mode (for example, using a thin layer for the file system that calls the user mode application). In one embodiment, we may prefer not to use this approach, because it can cause system lock-ups even in normal conditions.

A kernel file system – in accordance with some embodiments of the invention, this implementation may be similar to the one implemented in Linux.

Virtual Machine installed – some embodiments may allow to install a virtual machine (such as VMWare, Plex86, Virtual-PC and other suitable products) on the client, install Linux OS on top, and install the FileCache™ on top. Then, configure the virtual machine to be a host in the network, and make sure it is part of the user's domain. Now all the user should do is use the system in the same way he would use it as if it were a separate appliance. This way, little implementation is needed and almost no user training. However, there may be a performance

consideration – since VMWare solutions tend to use more CPU and memory than just the FileCache™ application itself.

6. Glossary

The following terms may include, for example, the content as described in the corresponding definitions, as follows:

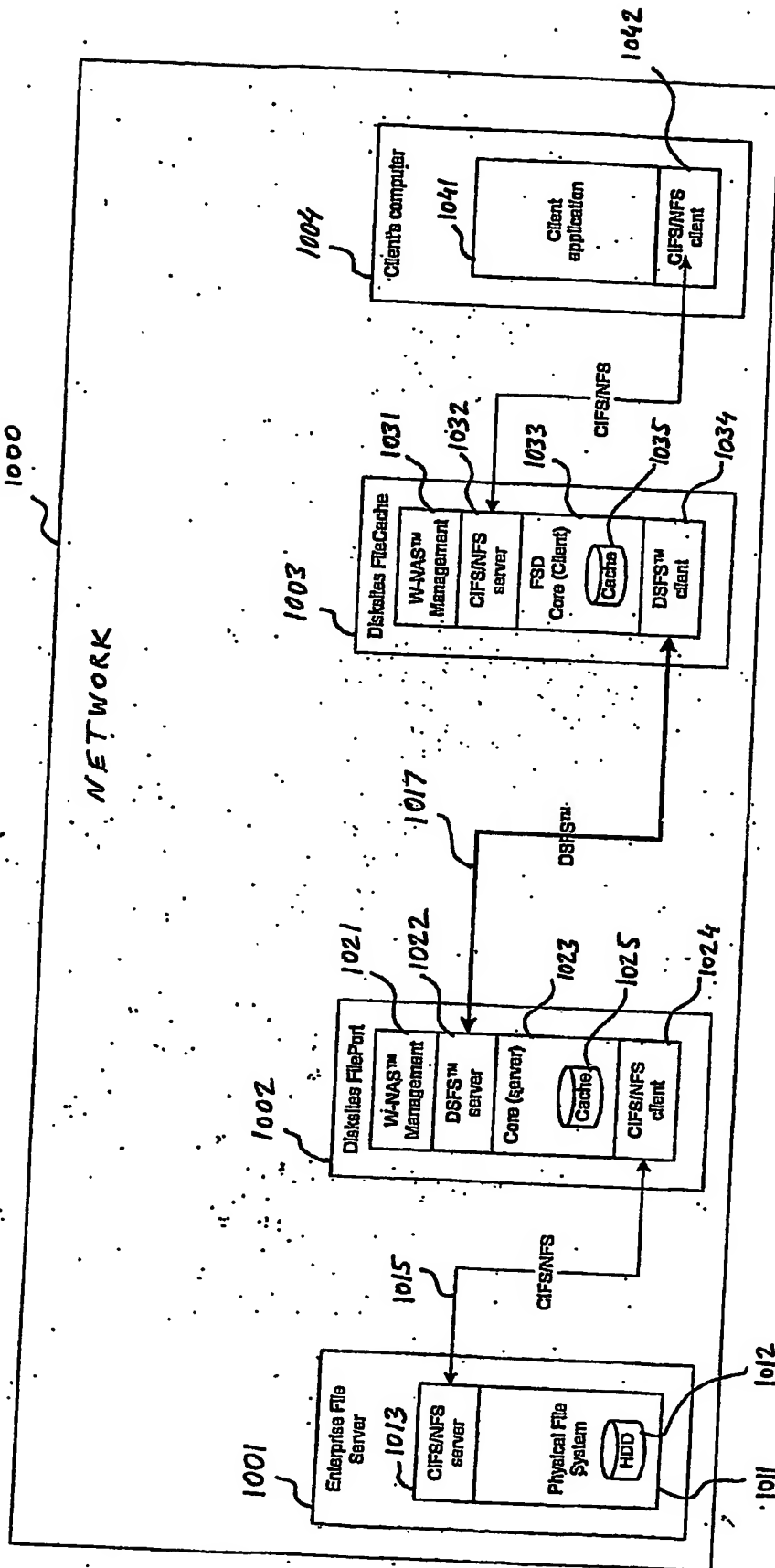
- Op-cache - Operations' result security cache. This cache has the form <user, verb, object, result>. For instance, it knows that user A, tried to OPEN a file X, and got ACCESS DENIED. It doesn't know any other information about the user or the file's ACL.
- EFS - Enterprise File Server
- Latency - The time it takes a packet to go back and forth between two machines on a network. It is sometimes called "roundtrip latency" or "ping time"
- FileCache™ - DiskSites' appliance, installed at the remote branch, and acts as a file server while redirecting all requests to the FilePort™.
- FilePort™ - DiskSites' appliance, installed at the center. It receives all requests from the FileCache™, and performs them on the EFS, on behalf of the original user.
- TTL - Time to live. Each cached piece of information has a period of time, within which it is considered valid. The time to live is that amount of time
- ACL - Access Control List. For each object in a file system, the administrator maintains an ACL. It contains information about who is entitled to do what operations on the file.
- TCP - Transport Control Protocol.
- IP - Internet Protocol
- CIFS - Common Internet File System
- NFS - Network File System
- SD - Security Descriptor
- Hash - a relatively fast algorithm to digitally sign a stream of data. Well known hashes include SHA, MD4, MD5. It is characterized by the fact that there is no easy method to compute the stream of data from the hash (besides trying all the possibilities until a hit).
- MD5 - a hash algorithm

- Commit operation – This is the operation in our protocol that sends the new written data to the FilePort™. Data sent can be any of full file, diff, or other method of marking the file data to the other side.

Claims

1. A method substantially as shown and described hereinabove and/or with reference to any of the accompanying drawings.
2. A system substantially as shown and described hereinabove and/or with reference to any of the accompanying drawings.

FIG. 1



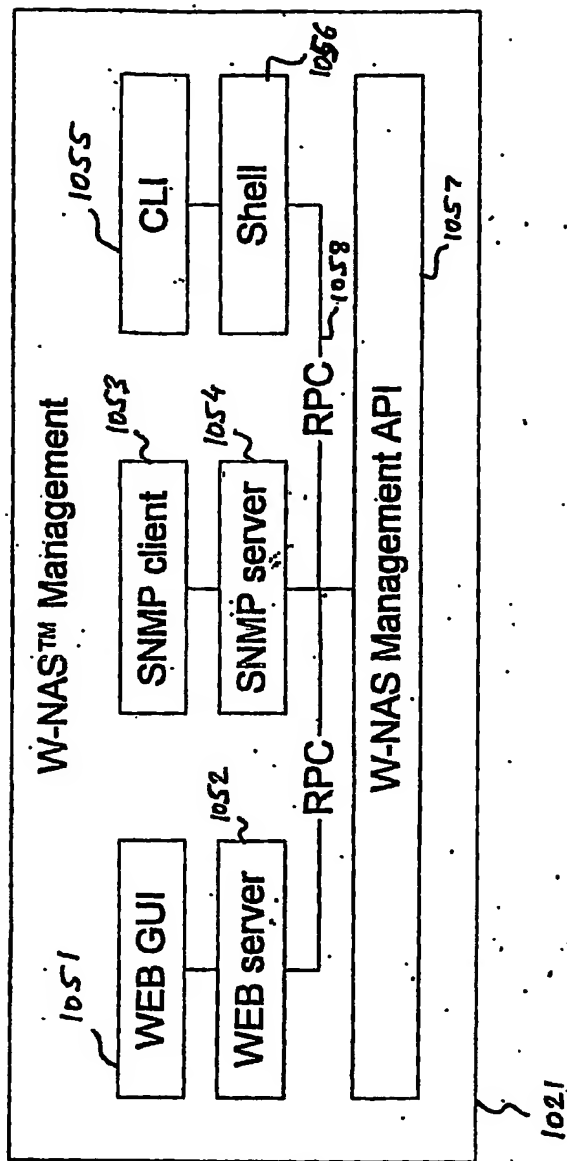


FIG. 2